ESMValTool User's and Developer's Guide

Release 2.3.0

ESMValTool Development Team

Jun 14, 2021

ESMVALTOOL

Part I

Getting started

CHAPTER

ONE

INSTALLATION

1.1 Conda installation

In order to install the Conda package, you will need to install Conda first. For a minimal conda installation (recommended) go to https://conda.io/miniconda.html. It is recommended that you always use the latest version of conda, as problems have been reported when trying to use older versions.

Once you have installed conda, you can install ESMValCore by running:

```
conda install -c esmvalgroup -c conda-forge esmvalcore
```

It is also possible to create a new Conda environment and install ESMValCore into it with a single command:

```
conda create --name esmvalcore -c esmvalgroup -c conda-forge esmvalcore
```

Don't forget to activate the newly created environment after the installation:

```
conda activate esmvalcore
```

Of course it is also possible to choose a different name than `esmvalcore` for the environment.

Note: Creating a new Conda environment is often much faster and more reliable than trying to update an existing Conda environment.

1.2 Pip installation

It is also possible to install ESMValCore from PyPI. However, this requires first installing dependencies that are not available on PyPI in some other way. By far the easiest way to install these dependencies is to use conda. For a minimal conda installation (recommended) go to https://conda.io/miniconda.html.

After installing Conda, download the file with the list of dependencies:

```
wget https://raw.githubusercontent.com/ESMValGroup/ESMValCore/main/environment.yml
```

and install these dependencies into a new conda environment with the command

```
conda env create --name esmvalcore --file environment.yml
```

Finally, activate the newly created environment

```
conda activate esmvalcore
```

and install ESMValCore as well as any remaining dependencies with the command:

```
pip install esmvalcore
```

1.3 Docker installation

ESMValCore is also provided through DockerHub in the form of docker containers. See https://docs.docker.com for more information about docker containers and how to run them.

You can get the latest release with

```
docker pull esmvalgroup/esmvalcore:stable
```

If you want to use the current main branch, use

```
docker pull esmvalgroup/esmvalcore:latest
```

To run a container using those images, use:

```
docker run esmvalgroup/esmvalcore:stable --help
```

Note that the container does not see the data or environmental variables available in the host by default. You can make data available with `-v /path:/path/in/container` and environmental variables with `-e VARNAME`.

For example, the following command would run a recipe

```
docker run -e HOME -v "$HOME":"$HOME" -v /data:/data esmvalgroup/esmvalcore:stable -c ~/
↪config-user.yml ~/recipes/recipe_example.yml
```

with the environmental variable `$HOME` available inside the container and the data in the directories `$HOME` and `/data`, so these can be used to find the configuration file, recipe, and data.

It might be useful to define a bash alias or script to abbreviate the above command, for example

```
alias esmvaltool="docker run -e HOME -v $HOME:$HOME -v /data:/data esmvalgroup/
↪esmvalcore:stable"
```

would allow using the `esmvaltool` command without even noticing that the tool is running inside a Docker container.

1.4 Singularity installation

Docker is usually forbidden in clusters due to security reasons. However, there is a more secure alternative to run containers that is usually available on them: Singularity.

Singularity can use docker containers directly from DockerHub with the following command

```
singularity run docker://esmvalgroup/esmvalcore:stable -c ~/config-user.yml ~/recipes/
↪recipe_example.yml
```

Note that the container does not see the data available in the host by default. You can make host data available with `-B /path:/path/in/container`.

It might be useful to define a bash alias or script to abbreviate the above command, for example

```
alias esmvaltool="singularity run -B $HOME:$HOME -B /data:/data docker://esmvalgroup/
↪esmvalcore:stable"
```

would allow using the `esmvaltool` command without even noticing that the tool is running inside a Singularity container.

Some clusters may not allow to connect to external services, in those cases you can first create a singularity image locally:

```
singularity build esmvalcore.sif docker://esmvalgroup/esmvalcore:stable
```

and then upload the image file `esmvalcore.sif` to the cluster. To run the container using the image file `esmvalcore.sif` use:

```
singularity run esmvalcore.sif -c ~/config-user.yml ~/recipes/recipe_example.yml
```

1.5 Installation from source

Note: If you would like to install the development version of ESMValCore alongside ESMValTool, please have a look at these instructions.

To install from source for development, follow these instructions.

- Download and install conda (this should be done even if the system in use already has a preinstalled version of conda, as problems have been reported with using older versions of conda)
- To make the `conda` command available, add `source <prefix>/etc/profile.d/conda.sh` to your `.bashrc` file and restart your shell. If using (t)csh shell, add `source <prefix>/etc/profile.d/conda.csh` to your `.cshrc/.tcshrc` file instead.
- Update conda: `conda update -y conda`
- Clone the ESMValCore Git repository: `git clone https://github.com/ESMValGroup/ESMValCore.git`
- Go to the source code directory: `cd ESMValCore`
- Create the esmvalcore conda environment `conda env create --name esmvalcore --file environment.yml`
- Activate the esmvalcore environment: `conda activate esmvalcore`
- Install in development mode: `pip install -e '.[develop]'`. If you are installing behind a proxy that does not trust the usual pip-urls you can declare them with the option `--trusted-host`, e.g. `pip install --trusted-host=pypi.python.org --trusted-host=pypi.org --trusted-host=files.pythonhosted.org -e .[develop]`
- Test that your installation was successful by running `esmvaltool -h`.

1.6 Pre-installed versions on HPC clusters

You will find the tool available on HPC clusters and there will be no need to install it yourself if you are just running diagnostics:

- CEDA-JASMIN: *esmvaltool* is available on the scientific compute nodes (*sciX.jasmin.ac.uk* where *X = 1, 2, 3, 4, 5*) after login and module loading via *module load esmvaltool*; see the helper page at CEDA ;
- DKRZ-Mistral: *esmvaltool* is available on login nodes (*mistral.dkrz.de*) and pre- and post-processing nodes (*mistralpp.dkrz.de*) after login and module loading via *module load esmvaltool*; the command *module help esmvaltool* provides some information about the module.

CHAPTER

TWO

CONFIGURATION FILES

2.1 Overview

There are several configuration files in ESMValCore:

- `config-user.yml`: sets a number of user-specific options like desired graphical output format, root paths to data, etc.;
- `config-developer.yml`: sets a number of standardized file-naming and paths to data formatting;

and one configuration file which is distributed with ESMValTool:

- `config-references.yml`: stores information on diagnostic and recipe authors and scientific journals references;

2.2 User configuration file

The `config-user.yml` configuration file contains all the global level information needed by ESMValTool. It can be reused as many times the user needs to before changing any of the options stored in it. This file is essentially the gateway between the user and the machine-specific instructions to `esmvaltool`. By default, esmvaltool looks for it in the home directory, inside the `.esmvaltool` folder.

Users can get a copy of this file with default values by running

```
esmvaltool config get-config-user --path=${TARGET_FOLDER}
```

If the option `--path` is omitted, the file will be created in `${HOME}/.esmvaltool`

The following shows the default settings from the `config-user.yml` file with explanations in a commented line above each option:

```
# Set the console log level debug, [info], warning, error
# for much more information printed to screen set log_level: debug
log_level: info

# Exit on warning (only for NCL diagnostic scripts)? true/[false]
exit_on_warning: false

# Plot file format? [png]/pdf/ps/eps/epsi
output_file_type: png

# Destination directory where all output will be written
```

(continues on next page)

(continued from previous page)

```yaml
# including log files and performance stats
output_dir: ./esmvaltool_output

# Auxiliary data directory (used for some additional datasets)
# this is where e.g. files can be downloaded to by a download
# script embedded in the diagnostic
auxiliary_data_dir: ./auxiliary_data

# Use netCDF compression true/[false]
compress_netcdf: false

# Save intermediary cubes in the preprocessor true/[false]
# set to true will save the output cube from each preprocessing step
# these files are numbered according to the preprocessing order
save_intermediary_cubes: false

# Remove the preproc dir if all fine
# if this option is set to "true", ALL preprocessor files will be removed
# CAUTION when using: if you need those files, set it to false
remove_preproc_dir: true

# Run at most this many tasks in parallel [null]/1/2/3/4/..
# Set to null to use the number of available CPUs.
# If you run out of memory, try setting max_parallel_tasks to 1 and check the
# amount of memory you need for that by inspecting the file
# run/resource_usage.txt in the output directory. Using the number there you
# can increase the number of parallel tasks again to a reasonable number for
# the amount of memory available in your system.
max_parallel_tasks: null

# Path to custom config-developer file, to customise project configurations.
# See config-developer.yml for an example. Set to None to use the default
config_developer_file: null

# Use a profiling tool for the diagnostic run [false]/true
# A profiler tells you which functions in your code take most time to run.
# For this purpose we use vprof, see below for notes
# Only available for Python diagnostics
profile_diagnostic: false

# Rootpaths to the data from different projects (lists are also possible)
rootpath:
  CMIP5: [~/cmip5_inputpath1, ~/cmip5_inputpath2]
  OBS: ~/obs_inputpath
  default: ~/default_inputpath

# Directory structure for input data: [default]/BADC/DKRZ/ETHZ/etc
# See config-developer.yml for definitions.
drs:
  CMIP5: default
```

There used to be a setting `write_plots` and `write_netcdf` in the config user file, but these have been deprecated since ESMValCore v2.2 and will be removed in v2.4, because only some diagnostic scripts supported these settings.

For those diagnostic scripts that do support these settings, they can now be configured in the diagnostic script section of the recipe.

```
# Auxiliary data directory (used for some additional datasets)
auxiliary_data_dir: ~/auxiliary_data
```

The `auxiliary_data_dir` setting is the path to place any required additional auxiliary data files. This is necessary because certain Python toolkits, such as cartopy, will attempt to download data files at run time, typically geographic data files such as coastlines or land surface maps. This can fail if the machine does not have access to the wider internet. This location allows the user to specify where to find such files if they can not be downloaded at runtime.

Warning: This setting is not for model or observational datasets, rather it is for data files used in plotting such as coastline descriptions and so on.

The `profile_diagnostic` setting triggers profiling of Python diagnostics, this will tell you which functions in the diagnostic took most time to run. For this purpose we use vprof. For each diagnostic script in the recipe, the profiler writes a `.json` file that can be used to plot a flame graph of the profiling information by running

```
vprof --input-file esmvaltool_output/recipe_output/run/diagnostic/script/profile.json
```

Note that it is also possible to use vprof to understand other resources used while running the diagnostic, including execution time of different code blocks and memory usage.

A detailed explanation of the data finding-related sections of the `config-user.yml` (`rootpath` and `drs`) is presented in the *Data retrieval* section. This section relates directly to the data finding capabilities of ESMValTool and are very important to be understood by the user.

Note: You can choose your `config-user.yml` file at run time, so you could have several of them available with different purposes. One for a formalised run, another for debugging, etc. You can even provide any config user value as a run flag `--argument_name argument_value`

2.3 Developer configuration file

Most users and diagnostic developers will not need to change this file, but it may be useful to understand its content. It will be installed along with ESMValCore and can also be viewed on GitHub: esmvalcore/config-developer.yml. This configuration file describes the file system structure and CMOR tables for several key projects (CMIP6, CMIP5, obs4mips, OBS6, OBS) on several key machines (e.g. BADC, CP4CDS, DKRZ, ETHZ, SMHI, BSC), and for native output data for some models (IPSL, … see *Configuring native models and observation data sets*). CMIP data is stored as part of the Earth System Grid Federation (ESGF) and the standards for file naming and paths to files are set out by CMOR and DRS. For a detailed description of these standards and their adoption in ESMValCore, we refer the user to *CMIP data* section where we relate these standards to the data retrieval mechanism of the ESMValCore.

By default, esmvaltool looks for it in the home directory, inside the '.esmvaltool' folder.

Users can get a copy of this file with default values by running

```
esmvaltool config get-config-developer --path=${TARGET_FOLDER}
```

If the option `--path` is omitted, the file will be created in `` `${HOME}/.esmvaltool ``.

Note: Remember to change your config-user file if you want to use a custom config-developer.

Example of the CMIP6 project configuration:

```
CMIP6:
  input_dir:
    default: '/'
    BADC: '{activity}/{institute}/{dataset}/{exp}/{ensemble}/{mip}/{short_name}/{grid}/
↪{latestversion}'
    DKRZ: '{activity}/{institute}/{dataset}/{exp}/{ensemble}/{mip}/{short_name}/{grid}/
↪{latestversion}'
    ETHZ: '{exp}/{mip}/{short_name}/{dataset}/{ensemble}/{grid}/'
  input_file: '{short_name}_{mip}_{dataset}_{exp}_{ensemble}_{grid}*.nc'
  output_file: '{project}_{dataset}_{mip}_{exp}_{ensemble}_{short_name}'
  cmor_type: 'CMIP6'
  cmor_strict: true
```

2.3.1 Input file paths

When looking for input files, the `esmvaltool` command provided by ESMValCore replaces the placeholders `{item}` in `input_dir` and `input_file` with the values supplied in the recipe. ESMValCore will try to automatically fill in the values for institute, frequency, and modeling_realm based on the information provided in the CMOR tables and/or `config-developer.yml` when reading the recipe. If this fails for some reason, these values can be provided in the recipe too.

The data directory structure of the CMIP projects is set up differently at each site. As an example, the CMIP6 directory path on BADC would be:

```
'{activity}/{institute}/{dataset}/{exp}/{ensemble}/{mip}/{short_name}/{grid}/
↪{latestversion}'
```

The resulting directory path would look something like this:

```
CMIP/MOHC/HadGEM3-GC31-LL/historical/r1i1p1f3/Omon/tos/gn/latest
```

Please, bear in mind that `input_dirs` can also be a list for those cases in which may be needed:

```
- '{exp}/{ensemble}/original/{mip}/{short_name}/{grid}/{latestversion}'
- '{exp}/{ensemble}/computed/{mip}/{short_name}/{grid}/{latestversion}'
```

In that case, the resultant directories will be:

```
historical/r1i1p1f3/original/Omon/tos/gn/latest
historical/r1i1p1f3/computed/Omon/tos/gn/latest
```

For a more in-depth description of how to configure ESMValCore so it can find your data please see *CMIP data*.

2.3.2 Preprocessor output files

The filename to use for preprocessed data is configured in a similar manner using `output_file`. Note that the extension `.nc` (and if applicable, a start and end time) will automatically be appended to the filename.

2.3.3 Project CMOR table configuration

ESMValCore comes bundled with several CMOR tables, which are stored in the directory esmvalcore/cmor/tables. These are copies of the tables available from PCMDI.

For every `project` that can be used in the recipe, there are four settings related to CMOR table settings available:

- `cmor_type`: can be `CMIP5` if the CMOR table is in the same format as the CMIP5 table or `CMIP6` if the table is in the same format as the CMIP6 table.
- `cmor_strict`: if this is set to `false`, the CMOR table will be extended with variables from the `esmvalcore/cmor/tables/custom` directory and it is possible to use variables with a `mip` which is different from the MIP table in which they are defined.
- `cmor_path`: path to the CMOR table. Relative paths are with respect to esmvalcore/cmor/tables. Defaults to the value provided in `cmor_type` written in lower case.
- `cmor_default_table_prefix`: Prefix that needs to be added to the `mip` to get the name of the file containing the `mip` table. Defaults to the value provided in `cmor_type`.

2.3.4 Configuring native models and observation data sets

ESMValCore can be configured for handling native model output formats and specific observation data sets without preliminary reformatting. You can choose to host this new data source either under a dedicated project or under project `native6`; when choosing the latter, such a configuration involves the following steps:

- allowing for ESMValTool to locate the data files:
  - entry `native6` of `config-developer.yml` should be complemented with sub-entries for `input_dir` and `input_file` that goes under a new key representing the data organization (such as `MY_DATA_ORG`), and these sub-entries can use an arbitrary list of `{placeholders}`. Example :

```
native6:
  ...
  input_dir:
     default: 'Tier{tier}/{dataset}/{latestversion}/{frequency}/{short_name}'
     MY_DATA_ORG: '{model}/{exp}/{simulation}/{version}/{type}'
  input_file:
    default: '*.nc'
    MY_DATA_ORG: '{simulation}_*.nc'
  ...
```

  - if necessary, provide a so-called `extra facets file` which allows to cope e.g. with variable naming issues for finding files. See *Extra Facets* and `this example of such a file for IPSL-CM6`.
- ensuring that ESMValCore get the right metadata and data out of your data files: this is described in *Fixing data*

2.4 References configuration file

The esmvaltool/config-references.yml file contains the list of ESMValTool diagnostic and recipe authors, references and projects. Each author, project and reference referred to in the documentation section of a recipe needs to be in this file in the relevant section.

For instance, the recipe `recipe_ocean_example.yml` file contains the following documentation section:

```
documentation:
  authors:
    - demo_le

  maintainer:
    - demo_le

  references:
    - demora2018gmd

  projects:
    - ukesm
```

These four items here are named people, references and projects listed in the `config-references.yml` file.

2.5 Extra Facets

Sometimes it is useful to provide extra information for the loading of data, particularly in the case of native model data, or observational or other data, that generally follows the established standards, but is not part of the big supported projects like CMIP, CORDEX, obs4MIPs.

To support this, we provide the extra facets facilities. Facets are the key-value pairs described in *Recipe section: datasets*. Extra facets allows for the addition of more details per project, dataset, mip table, and variable name.

More precisely, one can provide this information in an extra yaml file, named *{project}-something.yml*, where *{project}* corresponds to the project as used by ESMValTool in *Recipe section: datasets* and "something" is arbitrary.

2.5.1 Format of the extra facets files

The extra facets are given in a yaml file, whose file name identifies the project. Inside the file there is a hierarchy of nested dictionaries with the following levels. At the top there is the *dataset* facet, followed by the *mip* table, and finally the *short_name*. The leaf dictionary placed here gives the extra facets that will be made available to data finder and the fix infrastructure. The following example illustrates the concept.

Listing 1: Extra facet example file *native6-era5.yml*

```
ERA5:
  Amon:
    tas: {source_var_name: "t2m", cds_var_name: "2m_temperature"}
```

2.5.2 Location of the extra facets files

Extra facets files can be placed in several different places. When we use them to support a particular use-case within the ESMValTool project, they will be provided in the sub-folder *extra_facets* inside the package *esmvalcore._config*. If they are used from the user side, they can be either placed in *~/.esmvaltool/extra_facets* or in any other directory of the users choosing. In that case this directory must be added to the *config-user.yml* file under the *extra_facets_dir* setting, which can take a single directory or a list of directories.

The order in which the directories are searched is

1. The internal directory *esmvalcore._config/extra_facets*
2. The default user directory *~/.esmvaltool/extra_facets*
3. The custom user directories in the order in which they are given in *config-user.yml*.

The extra facets files within each of these directories are processed in lexicographical order according to their file name.

In all cases it is allowed to supersede information from earlier files in later files. This makes it possible for the user to effectively override even internal default facets, for example to deal with local particularities in the data handling.

2.5.3 Use of extra facets

For extra facets to be useful, the information that they provide must be applied. There are fundamentally two places where this comes into play. One is *the datafinder*, the other are *fixes*.

CHAPTER

THREE

INPUT DATA

3.1 Overview

Data discovery and retrieval is the first step in any evaluation process; ESMValTool uses a *semi-automated* data finding mechanism with inputs from both the user configuration file and the recipe file: this means that the user will have to provide the tool with a set of parameters related to the data needed and once these parameters have been provided, the tool will automatically find the right data. We will detail below the data finding and retrieval process and the input the user needs to specify, giving examples on how to use the data finding routine under different scenarios.

3.2 Data types

3.2.1 CMIP data

CMIP data is widely available via the Earth System Grid Federation (ESGF) and is accessible to users either via download from the ESGF portal or through the ESGF data nodes hosted by large computing facilities (like CEDA-Jasmin, DKRZ, etc). This data adheres to, among other standards, the DRS and Controlled Vocabulary standard for naming files and structured paths; the DRS ensures that files and paths to them are named according to a standardized convention. Examples of this convention, also used by ESMValTool for file discovery and data retrieval, include:

- CMIP6 file: `[variable_short_name]_[mip]_[dataset_name]_[experiment]_[ensemble]_[grid]_[start-date]-[e`
  `nc`
- CMIP5 file: `[variable_short_name]_[mip]_[dataset_name]_[experiment]_[ensemble]_[start-date]-[end-date`
  `nc`
- OBS file: `[project]_[dataset_name]_[type]_[version]_[mip]_[short_name]_[start-date]-[end-date].`
  `nc`

Similar standards exist for the standard paths (input directories); for the ESGF data nodes, these paths differ slightly, for example:

- CMIP6 path for BADC: `ROOT-BADC/[institute]/[dataset_name]/[experiment]/[ensemble]/[mip]/ [variable_short_name]/[grid]`;
- CMIP6 path for ETHZ: `ROOT-ETHZ/[experiment]/[mip]/[variable_short_name]/[dataset_name]/[ensemble]/[grid]`

From the ESMValTool user perspective the number of data input parameters is optimized to allow for ease of use. We detail this procedure in the next section.

3.2.2 Native model data

Support for native model data that is not formatted according to a CMIP data request is quite easy using basic *ESMValCore fix procedure* and has been implemented for some models *as described here*

3.2.3 Observational data

Part of observational data is retrieved in the same manner as CMIP data, for example using the `OBS` root path set to:

```
OBS: /gws/nopw/j04/esmeval/obsdata-v2
```

and the dataset:

```
- {dataset: ERA-Interim, project: OBS, type: reanaly, version: 1, start_year:␣
↪2014, end_year: 2015, tier: 3}
```

in `recipe.yml` in `datasets` or `additional_datasets`, the rules set in *CMOR-DRS* are used again and the file will be automatically found:

```
/gws/nopw/j04/esmeval/obsdata-v2/Tier3/ERA-Interim/OBS_ERA-Interim_reanaly_1_Amon_ta_
↪201401-201412.nc
```

Since observational data are organized in Tiers depending on their level of public availability, the `default` directory must be structured accordingly with sub-directories `TierX` (`Tier1`, `Tier2` or `Tier3`), even when `drs: default`.

3.3 Data retrieval

Data retrieval in ESMValTool has two main aspects from the user's point of view:

- data can be found by the tool, subject to availability on disk;
- it is the user's responsibility to set the correct data retrieval parameters;

The first point is self-explanatory: if the user runs the tool on a machine that has access to a data repository or multiple data repositories, then ESMValTool will look for and find the available data requested by the user.

The second point underlines the fact that the user has full control over what type and the amount of data is needed for the analyses. Setting the data retrieval parameters is explained below.

3.3.1 Setting the correct root paths

The first step towards providing ESMValTool the correct set of parameters for data retrieval is setting the root paths to the data. This is done in the user configuration file `config-user.yml`. The two sections where the user will set the paths are `rootpath` and `drs`. `rootpath` contains pointers to `CMIP`, `OBS`, `default` and `RAWOBS` root paths; `drs` sets the type of directory structure the root paths are structured by. It is important to first discuss the `drs` parameter: as we've seen in the previous section, the DRS as a standard is used for both file naming conventions and for directory structures.

3.3.2 Synda

If the synda install command is used to download data, it maintains the directory structure as on ESGF. To find data downloaded by synda, use the `SYNDA` `drs` parameter.

```
drs:
  CMIP6: SYNDA
  CMIP5: SYNDA
```

3.3.3 Explaining `config-user/drs: CMIP5:` or `config-user/drs: CMIP6:`

Whereas ESMValTool will **always** use the CMOR standard for file naming (please refer above), by setting the `drs` parameter the user tells the tool what type of root paths they need the data from, e.g.:

```
drs:
  CMIP6: BADC
```

will tell the tool that the user needs data from a repository structured according to the BADC DRS structure, i.e.:

`ROOT/[institute]/[dataset_name]/[experiment]/[ensemble]/[mip]/[variable_short_name]/[grid]`;

setting the `ROOT` parameter is explained below. This is a strictly-structured repository tree and if there are any sort of irregularities (e.g. there is no `[mip]` directory) the data will not be found! `BADC` can be replaced with `DKRZ` or `ETHZ` depending on the existing `ROOT` directory structure. The snippet

```
drs:
  CMIP6: default
```

is another way to retrieve data from a `ROOT` directory that has no DRS-like structure; `default` indicates that the data lies in a directory that contains all the files without any structure.

Note: When using `CMIP6: default` or `CMIP5: default` it is important to remember that all the needed files must be in the same top-level directory set by `default` (see below how to set `default`).

3.3.4 Explaining `config-user/rootpath:`

`rootpath` identifies the root directory for different data types (`ROOT` as we used it above):

- `CMIP` e.g. `CMIP5` or `CMIP6`: this is the *root* path(s) to where the CMIP files are stored; it can be a single path or a list of paths; it can point to an ESGF node or it can point to a user private repository. Example for a CMIP5 root path pointing to the ESGF node on CEDA-Jasmin (formerly known as BADC):

  ```
  CMIP5: /badc/cmip5/data/cmip5/output1
  ```

  Example for a CMIP6 root path pointing to the ESGF node on CEDA-Jasmin:

  ```
  CMIP6: /badc/cmip6/data/CMIP6/CMIP
  ```

  Example for a mix of CMIP6 root path pointing to the ESGF node on CEDA-Jasmin and a user-specific data repository for extra data:

```
CMIP6: [/badc/cmip6/data/CMIP6/CMIP, /home/users/johndoe/cmip_data]
```

- OBS: this is the *root* path(s) to where the observational datasets are stored; again, this could be a single path or a list of paths, just like for CMIP data. Example for the OBS path for a large cache of observation datasets on CEDA-Jasmin:

```
OBS: /gws/nopw/j04/esmeval/obsdata-v2
```

- default: this is the *root* path(s) to where files are stored without any DRS-like directory structure; in a nutshell, this is a single directory that should contain all the files needed by the run, without any sub-directory structure.
- RAWOBS: this is the *root* path(s) to where the raw observational data files are stored; this is used by cmorize_obs.

3.3.5 Dataset definitions in recipe

Once the correct paths have been established, ESMValTool collects the information on the specific datasets that are needed for the analysis. This information, together with the CMOR convention for naming files (see *CMOR-DRS*) will allow the tool to search and find the right files. The specific datasets are listed in any recipe, under either the datasets and/or additional_datasets sections, e.g.

```
datasets:
  - {dataset: HadGEM2-CC, project: CMIP5, exp: historical, ensemble: r1i1p1, start_year:␣
↪2001, end_year: 2004}
  - {dataset: UKESM1-0-LL, project: CMIP6, exp: historical, ensemble: r1i1p1f2, grid: gn,
↪ start_year: 2004, end_year: 2014}
```

_data_finder will use this information to find data for **all** the variables specified in diagnostics/variables.

3.4 Recap and example

Let us look at a practical example for a recap of the information above: suppose you are using a config-user.yml that has the following entries for data finding:

```
rootpath:  # running on CEDA-Jasmin
  CMIP6: /badc/cmip6/data/CMIP6/CMIP
drs:
  CMIP6: BADC  # since you are on CEDA-Jasmin
```

and the dataset you need is specified in your recipe.yml as:

```
- {dataset: UKESM1-0-LL, project: CMIP6, mip: Amon, exp: historical, grid: gn, ensemble:␣
↪r1i1p1f2, start_year: 2004, end_year: 2014}
```

for a variable, e.g.:

```
diagnostics:
  some_diagnostic:
    description: some_description
    variables:
      ta:
        preprocessor: some_preprocessor
```

The tool will then use the root path `/badc/cmip6/data/CMIP6/CMIP` and the dataset information and will assemble the full DRS path using information from *CMOR-DRS* and establish the path to the files as:

```
/badc/cmip6/data/CMIP6/CMIP/MOHC/UKESM1-0-LL/historical/r1i1p1f2/Amon
```

then look for variable `ta` and specifically the latest version of the data file:

```
/badc/cmip6/data/CMIP6/CMIP/MOHC/UKESM1-0-LL/historical/r1i1p1f2/Amon/ta/gn/latest/
```

and finally, using the file naming definition from *CMOR-DRS* find the file:

```
/badc/cmip6/data/CMIP6/CMIP/MOHC/UKESM1-0-LL/historical/r1i1p1f2/Amon/ta/gn/latest/ta_
↪Amon_UKESM1-0-LL_historical_r1i1p1f2_gn_195001-201412.nc
```

3.5 Data loading

Data loading is done using the data load functionality of *iris*; we will not go into too much detail about this since we can point the user to the specific functionality here but we will underline that the initial loading is done by adhering to the CF Conventions that *iris* operates by as well (see CF Conventions Document and the search page for CF standard names).

3.6 Data concatenation from multiple sources

Oftentimes data retrieving results in assembling a continuous data stream from multiple files or even, multiple experiments. The internal mechanism through which the assembly is done is via cube concatenation. One peculiarity of iris concatenation (see iris cube concatenation) is that it doesn't allow for concatenating time-overlapping cubes; this case is rather frequent with data from models overlapping in time, and is accounted for by a function that performs a flexible concatenation between two cubes, depending on the particular setup:

- cubes overlap in time: resulting cube is made up of the overlapping data plus left and right hand sides on each side of the overlapping data; note that in the case of the cubes coming from different experiments the resulting concatenated cube will have composite data made up from multiple experiments: assume [cube1: exp1, cube2: exp2] and cube1 starts before cube2, and cube2 finishes after cube1, then the concatenated cube will be made up of cube2: exp2 plus the section of cube1: exp1 that contains data not provided in cube2: exp2;
- cubes don't overlap in time: data from the two cubes is bolted together;

Note that two cube concatenation is the base operation of an iterative process of reducing multiple cubes from multiple data segments via cube concatenation ie if there is no time-overlapping data, the cubes concatenation is performed in one step.

3.7 Use of extra facets in the datafinder

Extra facets are a mechanism to provide additional information for certain kinds of data. The general approach is described in *Extra Facets*. Here, we describe how they can be used to locate data files within the datafinder framework. This is useful to build paths for directory structures and file names that follow a different system than the established DRS for, e.g. CMIP. A common application is the location of variables in multi-variable files as often found in climate models' native output formats.

Another use case is files that use different names for variables in their file name than for the netCDF4 variable name.

To apply the extra facets for this purpose, simply use the corresponding tag in the applicable DRS inside the *config-developer.yml* file. For example, given the extra facets in *Extra facet example file native6-era5.yml*, one might write the following.

Listing 1: Example drs use in *config-developer.yml*

```
native6:
  input_file:
    default: '{name_in_filename}*.nc'
```

The same replacement mechanism can be employed everywhere where tags can be used, particularly in *input_dir* and *input_file*.

CHAPTER

FOUR

WORKING WITH THE INSTALLED RECIPES

Although ESMValTool can be used just to simplify the management of data and the creation of your own analysis code, one of its main strengths is the continuously growing set of diagnostics and metrics that it directly provides to the user. These metrics and diagnostics are provided as a set of preconfigured recipes that users can run or customize for their own analysis. The latest list of available recipes can be found here.

In order to make the management of these installed recipes easier, ESMValTool provides the `recipes` command group with utilities that help the users in discovering and customizing the provided recipes.

The first command in this group allows users to get the complete list of installed recipes printed to the console:

```
esmvaltool recipes list
```

If the user then wants to explore any one of this recipes, they can be printed using the following command

```
esmvaltool recipes show recipe_name.yml
```

And finally, to get a local copy that can then be customized and run, users can use the following command

```
esmvaltool recipes get recipe_name.yml
```

CHAPTER

FIVE

RUNNING

The ESMValCore package provides the `esmvaltool` command line tool, which can be used to run a *recipe*.

To run a recipe, call `esmvaltool run` with the desired recipe:

```
esmvaltool run recipe_python.yml
```

If the configuration file is not in the default location `~/.esmvaltool/config-user.yml`, you can pass its path explicitly:

```
esmvaltool run --config_file /path/to/config-user.yml recipe_python.yml
```

It is also possible to explicitly change values from the config file using flags:

```
esmvaltool run --argument_name argument_value recipe_python.yml
```

To control the strictness of the CMOR checker, use the flag `--check_level`:

```
esmvaltool run --check_level=relaxed recipe_python.yml
```

Possible values are:

- *ignore*: all errors will be reported as warnings
- *relaxed*: only fail if there are critical errors
- *default*: fail if there are any errors
- *strict*: fail if there are any warnings

To run a reduced version of the recipe, usually for testing purpose you can use

```
esmvaltool run --max_datasets=NDATASETS --max_years=NYEARS recipe_python.yml
```

In this case, the recipe will limit the number of datasets per variable to NDATASETS and the total amount of years loaded to NYEARS. They can also be used separately.

To run a recipe, even if some datasets are not available, use

```
esmvaltool run --skip_nonexistent=True recipe_python.yml
```

If Synda is installed (see http://prodiguer.github.io/synda/), it is possible to use it to automatically download the requested data from ESGF if it is not available locally:

```
esmvaltool run --synda_download=True recipe_python.yml
```

It is also possible to select only specific diagnostics to be run. To tun only one, just specify its name. To provide more than one diagnostic to filter use the syntax 'diag1 diag2/script1' or '("diag1", "diag2/script1")' and pay attention to the quotes.

```
esmvaltool run --diagnostics=diagnostic1 recipe_python.yml
```

To get help on additional commands, please use

```
esmvaltool --help
```

Note: ESMValTool command line interface is created using the Fire python package. This package supports the creation of completion scripts for the Bash and Fish shells. Go to https://google.github.io/python-fire/using-cli/#python-fires-flags to learn how to set up them.

CHAPTER

SIX

OUTPUT

ESMValTool automatically generates a new output directory with every run. The location is determined by the output_dir option in the config-user.yml file, the recipe name, and the date and time, using the the format: `YYYYMMDD_HHMMSS`.

For instance, a typical output location would be: `output_directory/recipe_ocean_amoc_20190118_1027/`

This is effectively produced by the combination: `output_dir/recipe_name_YYYYMMDD_HHMMSS/`

This directory will contain 4 further subdirectories:

1. *Diagnostic output* (`work`): A place for any diagnostic script results that are not plots, e.g. files in NetCDF format (depends on the diagnostics).
2. *Plots* (`plots`): The location for all the plots, split by individual diagnostics and fields.
3. *Run* (`run`): This directory includes all log files, a copy of the recipe, a summary of the resource usage, and the *settings.yml* interface files and temporary files created by the diagnostic scripts.
4. *Preprocessed datasets* (`preproc`): This directory contains all the preprocessed netcdfs data and the *metadata.yml* interface files. Note that by default this directory will be deleted after each run, because most users will only need the results from the diagnostic scripts.

A summary of the output is produced in the file: `index.html`

6.1 Preprocessed datasets

The preprocessed datasets will be stored to the preproc/ directory. Each variable in each diagnostic will have its own the *metadata.yml* interface files saved in the preproc directory.

If the option `save_intermediary_cubes` is set to `true` in the config-user.yml file, then the intermediary cubes will also be saved here. This option is set to false in the default `config-user.yml` file.

If the option `remove_preproc_dir` is set to `true` in the config-user.yml file, then the preproc directory will be deleted after the run completes. This option is set to true in the default `config-user.yml` file.

6.2 Run

The log files in the run directory are automatically generated by ESMValTool and create a record of the output messages produced by ESMValTool and they are saved in the run directory. They can be helpful for debugging or monitoring the job, but also allow a record of the job output to screen after the job has been completed.

The run directory will also contain a copy of the recipe and the *settings.yml* file, described below. The run directory is also where the diagnostics are executed, and may also contain several temporary files while diagnostics are running.

6.3 Diagnostic output

The work/ directory will contain all files that are output at the diagnostic stage. Ie, the model data is preprocessed by ESMValTool and stored in the preproc/ directory. These files are opened by the diagnostic script, then some processing is applied. Once the diagnostic level processing has been applied, the results should be saved to the work directory.

6.4 Plots

The plots directory is where diagnostics save their output figures. These plots are saved in the format requested by the option *output_file_type* in the config-user.yml file.

6.5 Settings.yml

The settings.yml file is automatically generated by ESMValTool. Each diagnostic will produce a unique settings.yml file.

The settings.yml file passes several global level keys to diagnostic scripts. This includes several flags from the config-user.yml file (such as 'log_level'), several paths which are specific to the diagnostic being run (such as 'plot_dir' and 'run_dir') and the location on disk of the metadata.yml file (described below).

```
input_files:[[...]recipe_ocean_bgc_20190118_134855/preproc/diag_timeseries_scalars/mfo/
↪metadata.yml]
log_level: debug
output_file_type: png
plot_dir: [...]recipe_ocean_bgc_20190118_134855/plots/diag_timeseries_scalars/Scalar_
↪timeseries
profile_diagnostic: false
recipe: recipe_ocean_bgc.yml
run_dir: [...]recipe_ocean_bgc_20190118_134855/run/diag_timeseries_scalars/Scalar_
↪timeseries
script: Scalar_timeseries
version: 2.0a1
work_dir: [...]recipe_ocean_bgc_20190118_134855/work/diag_timeseries_scalars/Scalar_
↪timeseries
```

The first item in the settings file will be a list of *Metadata.yml* files. There is a metadata.yml file generated for each field in each diagnostic.

6.6 Metadata.yml

The metadata.yml files is automatically generated by ESMValTool. Along with the settings.yml file, it passes all the paths, boolean flags, and additional arguments that your diagnostic needs to know in order to run.

The metadata is loaded from cfg as a dictionairy object in python diagnostics.

Here is an example metadata.yml file:

```
?
  [...]/recipe_ocean_bgc_20190118_134855/preproc/diag_timeseries_scalars/mfo/CMIP5_
↪HadGEM2-ES_Omon_historical_r1i1p1_TO0M_mfo_2002-2004.nc
  : cmor_table: CMIP5
  dataset: HadGEM2-ES
  diagnostic: diag_timeseries_scalars
  end_year: 2004
  ensemble: r1i1p1
  exp: historical
  field: TO0M
  filename: [...]recipe_ocean_bgc_20190118_134855/preproc/diag_timeseries_scalars/mfo/
↪CMIP5_HadGEM2-ES_Omon_historical_r1i1p1_TO0M_mfo_2002-2004.nc
  frequency: mon
  institute: [INPE, MOHC]
  long_name: Sea Water Transport
  mip: Omon
  modeling_realm: [ocean]
  preprocessor: prep_timeseries_scalar
  project: CMIP5
  recipe_dataset_index: 0
  short_name: mfo
  standard_name: sea_water_transport_across_line
  start_year: 2002
  units: kg s-1
  variable_group: mfo
```

As you can see, this is effectively a dictionary with several items including data paths, metadata and other information.

There are several tools available in python which are built to read and parse these files. The tools are available in the shared directory in the diagnostics directory.

Part II

The recipe format

CHAPTER

SEVEN

OVERVIEW

After `config-user.yml`, the `recipe.yml` is the second file the user needs to pass to `esmvaltool` as command line option, at each run time point. Recipes contain the data and data analysis information and instructions needed to run the diagnostic(s), as well as specific diagnostic-related instructions.

Broadly, recipes contain a general section summarizing the provenance and functionality of the diagnostics, the datasets which need to be run, the preprocessors that need to be applied, and the diagnostics which need to be run over the preprocessed data. This information is provided to ESMValTool in four main recipe sections: *Documentation*, *Datasets*, *Preprocessors*, and *Diagnostics*, respectively.

7.1 Recipe section: `documentation`

The documentation section includes:

- The recipe's author's user name (`authors`, matching the definitions in the *References configuration file*)
- The recipe's maintainer's user name (`maintainer`, matching the definitions in the *References configuration file*)
- A description of the recipe (`description`, written in MarkDown format)
- A list of scientific references (`references`, matching the definitions in the *References configuration file*)
- the project or projects associated with the recipe (`projects`, matching the definitions in the *References configuration file*)

For example, the documentation section of `recipes/recipe_ocean_amoc.yml` is the following:

```
documentation:
  description: |
    Recipe to produce time series figures of the derived variable, the
    Atlantic meridional overturning circulation (AMOC).
    This recipe also produces transect figures of the stream functions for
    the years 2001-2004.

  authors:
    - demo_le

  maintainer:
    - demo_le

  references:
    - demora2018gmd
```

(continues on next page)

(continued from previous page)

```
  projects:
    - ukesm
```

Note: Note that all authors, projects, and references mentioned in the description section of the recipe need to be included in the (locally installed copy of the) file esmvaltool/config-references.yml, see *References configuration file*. The author name uses the format: `surname_name`. For instance, John Doe would be: `doe_john`. This information can be omitted by new users whose name is not yet included in `config-references.yml`.

7.2 Recipe section: `datasets`

The `datasets` section includes dictionaries that, via key-value pairs, define standardized data specifications:

- dataset name (key `dataset`, value e.g. `MPI-ESM-LR` or `UKESM1-0-LL`)
- project (key `project`, value `CMIP5` or `CMIP6` for CMIP data, `OBS` for observational data, `ana4mips` for ana4mips data, `obs4mips` for obs4mips data, `EMAC` for EMAC data)
- experiment (key `exp`, value e.g. `historical`, `amip`, `piControl`, `RCP8.5`)
- mip (for CMIP data, key `mip`, value e.g. `Amon`, `Omon`, `LImon`)
- ensemble member (key `ensemble`, value e.g. `r1i1p1`, `r1i1p1f1`)
- sub-experiment id (key *sub_experiment*, value e.g. *s2000*, *s(2000:2002)*, for DCPP data only)
- time range (e.g. key-value `start_year: 1982`, `end_year: 1990`. Please note that yaml interprets numbers with a leading `0` as octal numbers, so we recommend to avoid them. For example, use `128` to specify the year 128 instead of `0128`.)
- model grid (native grid `grid: gn` or regridded grid `grid: gr`, for CMIP6 data only).

For example, a datasets section could be:

```
datasets:
  - {dataset: CanESM2, project: CMIP5, exp: historical, ensemble: r1i1p1, start_year:␣
↪2001, end_year: 2004}
  - {dataset: UKESM1-0-LL, project: CMIP6, exp: historical, ensemble: r1i1p1f2, start_
↪year: 2001, end_year: 2004, grid: gn}
  - {dataset: EC-EARTH3, alias: custom_alias, project: CMIP6, exp: historical, ensemble:␣
↪r1i1p1f1, start_year: 2001, end_year: 2004, grid: gn}
  - {dataset: HadGEM3-GC31-MM, alias: custom_alias, project: CMIP6, exp: dcppA-hindcast,␣
↪ensemble: r1i1p1f1, sub_experiment: s2000, grid: gn, start_year: 2000, end_year, 2002}
```

It is possible to define the experiment as a list to concatenate two experiments. Here it is an example concatenating the *historical* experiment with *rcp85*

```
datasets:
  - {dataset: CanESM2, project: CMIP5, exp: [historical, rcp85], ensemble: r1i1p1, start_
↪year: 2001, end_year: 2004}
```

It is also possible to define the ensemble as a list when the two experiments have different ensemble names. In this case, the specified datasets are concatenated into a single cube:

```
datasets:
  - {dataset: CanESM2, project: CMIP5, exp: [historical, rcp85], ensemble: [r1i1p1,␣
↪r1i2p1], start_year: 2001, end_year: 2004}
```

ESMValTool also supports a simplified syntax to add multiple ensemble members from the same dataset. In the ensemble key, any element in the form *(x:y)* will be replaced with all numbers from x to y (both inclusive), adding a dataset entry for each replacement. For example, to add ensemble members r1i1p1 to r10i1p1 you can use the following abbreviated syntax:

```
datasets:
  - {dataset: CanESM2, project: CMIP5, exp: historical, ensemble: "r(1:10)i1p1", start_
↪year: 2001, end_year: 2004}
```

It can be included multiple times in one definition. For example, to generate the datasets definitions for the ensemble members r1i1p1 to r5i1p1 and from r1i2p1 to r5i1p1 you can use:

```
datasets:
  - {dataset: CanESM2, project: CMIP5, exp: historical, ensemble: "r(1:5)i(1:2)p1",␣
↪start_year: 2001, end_year: 2004}
```

Please, bear in mind that this syntax can only be used in the ensemble tag. Also, note that the combination of multiple experiments and ensembles, like exp: [historical, rcp85], ensemble: [r1i1p1, "r(2:3)i1p1"] is not supported and will raise an error.

The same simplified syntax can be used to add multiple sub-experiment ids:

```
datasets:
  - {dataset: MIROC6, project: CMIP6, exp: dcppA-hindcast, ensemble: r1i1p1f1, sub_
↪experiment: s(2000:2002), grid: gn, start_year: 2003, end_year: 2004}
```

Note that this section is not required, as datasets can also be provided in the *Diagnostics* section.

7.3 Recipe section: `preprocessors`

The preprocessor section of the recipe includes one or more preprocessors, each of which may call the execution of one or several preprocessor functions.

Each preprocessor section includes:

- A preprocessor name (any name, under `preprocessors`);
- A list of preprocessor steps to be executed (choose from the API);
- Any or none arguments given to the preprocessor steps;
- The order that the preprocessor steps are applied can also be specified using the `custom_order` preprocessor function.

The following snippet is an example of a preprocessor named `prep_map` that contains multiple preprocessing steps (*Horizontal regridding* with two arguments, *Time manipulation* with no arguments (i.e., calculating the average over the time dimension) and *Multi-model statistics* with two arguments):

```
preprocessors:
  prep_map:
    regrid:
```

(continues on next page)

(continued from previous page)

```
      target_grid: 1x1
      scheme: linear
    climate_statistics:
      operator: mean
    multi_model_statistics:
      span: overlap
      statistics: [mean]
```

Note: In this case no `preprocessors` section is needed the workflow will apply a `default` preprocessor consisting of only basic operations like: loading data, applying CMOR checks and fixes (*CMORization and dataset-specific fixes*) and saving the data to disk.

Preprocessor operations will be applied using the default order as listed in *Preprocessor functions*. Preprocessor tasks can be set to run in the order they are listed in the recipe by adding `custom_order: true` to the preprocessor definition.

7.4 Recipe section: `diagnostics`

The diagnostics section includes one or more diagnostics. Each diagnostic section will include:

- the variable(s) to preprocess, including the preprocessor to be applied to each variable;
- the diagnostic script(s) to be run;
- a description of the diagnostic and lists of themes and realms that it applies to;
- an optional `additional_datasets` section.

7.4.1 The diagnostics section defines tasks

The diagnostic section(s) define the tasks that will be executed when running the recipe. For each variable a preprocessing task will be defined and for each diagnostic script a diagnostic task will be defined. If variables need to be derived from other variables, a preprocessing task for each of the variables needed to derive that variable will be defined as well. These tasks can be viewed in the main_log_debug.txt file that is produced every run. Each task has a unique name that defines the subdirectory where the results of that task are stored. Task names start with the name of the diagnostic section followed by a '/' and then the name of the variable section for a preprocessing task or the name of the diagnostic script section for a diagnostic task.

A (simplified) example diagnostics section could look like

```
diagnostics:
  diagnostic_name:
    description: Air temperature tutorial diagnostic.
    themes:
      - phys
    realms:
      - atmos
    variables:
      variable_name:
        short_name: ta
        preprocessor: preprocessor_name
```

(continues on next page)

(continued from previous page)

```
      mip: Amon
    scripts:
      script_name:
        script: examples/diagnostic.py
```

Note that the example recipe above contains a single diagnostic section called `diagnostic_name` and will result in two tasks:

- a preprocessing task called `diagnostic_name/variable_name` that will preprocess air temperature data for each dataset in the *Datasets* section of the recipe (not shown).
- a diagnostic task called `diagnostic_name/script_name`

The path to the script provided in the `script` option should be either the absolute path to the script, or the path relative to the `esmvaltool/diag_scripts` directory.

Depending on the installation configuration, you may get an error of "file does not exist" when the system tries to run the diagnostic script using relative paths. If this happens, use an absolute path instead.

Note that the script should either have the extension for a supported language, i.e. `.py`, `.R`, `.ncl`, or `.jl` for Python, R, NCL, and Julia diagnostics respectively, or be executable if it is written in any other language.

7.4.2 Ancestor tasks

Some tasks require the result of other tasks to be ready before they can start, e.g. a diagnostic script needs the preprocessed variable data to start. Thus each tasks has zero or more ancestor tasks. By default, each diagnostic task in a diagnostic section has all variable preprocessing tasks in that same section as ancestors. However, this can be changed using the `ancestors` keyword. Note that wildcard expansion can be used to define ancestors.

```
diagnostics:
  diagnostic_1:
    variables:
      airtemp:
        short_name: ta
        preprocessor: preprocessor_name
        mip: Amon
    scripts:
      script_a:
        script: diagnostic_a.py
  diagnostic_2:
    variables:
      precip:
        short_name: pr
        preprocessor: preprocessor_name
        mip: Amon
    scripts:
      script_b:
        script: diagnostic_b.py
        ancestors: [diagnostic_1/script_a, precip]
```

The example recipe above will result in four tasks:

- a preprocessing task called `diagnostic_1/airtemp`
- a diagnostic task called `diagnostic_1/script_a`

- a preprocessing task called `diagnostic_2/precip`
- a diagnostic task called `diagnostic_2/script_b`

the preprocessing tasks do not have any ancestors, while the diagnostic_a.py script will receive the preprocessed air temperature data (has ancestor `diagnostic_1/airtemp`) and the diagnostic_b.py script will receive the results of diagnostic_a.py and the preprocessed precipitation data (has ancestors `diagnostic_1/script_a` and `diagnostic_2/precip`).

7.4.3 Task priority

Tasks are assigned a priority, with tasks appearing earlier on in the recipe getting higher priority. The tasks will be executed sequentially or in parellel, depending on the setting of `max_parallel_tasks` in the *User configuration file*. When there are fewer than `max_parallel_tasks` running, tasks will be started according to their priority. For obvious reasons, only tasks that are not waiting for ancestor tasks can be started. This feature makes it possible to reduce the processing time of recipes with many tasks, by placing tasks that take relatively long near the top of the recipe. Of course this only works when settings `max_parallel_tasks` to a value larger than 1. The current priority and run time of individual tasks can be seen in the log messages shown when running the tool (a lower number means higher priority).

7.4.4 Variable and dataset definitions

To define a variable/dataset combination that corresponds to an actual variable from a dataset, the keys in each variable section are combined with the keys of each dataset definition. If two versions of the same key are provided, then the key in the datasets section will take precedence over the keys in variables section. For many recipes it makes more sense to define the `start_year` and `end_year` items in the variable section, because the diagnostic script assumes that all the data has the same time range.

Variable short names usually do not change between datasets supported by ESMValCore, as they are usually changed to match CMIP. Nevertheless, there are small changes in variable names in CMIP6 with respect to CMIP5 (i.e. sea ice concentration changed from `sic` to `siconc`). ESMValCore is aware of some of them and can do the automatic translation when needed. It will even do the translation in the preprocessed file so the diagnostic does not have to deal with this complexity, setting the short name in all files to match the one used by the recipe. For example, if `sic` is requested, ESMValCore will find `sic` or `siconc` depending on the project, but all preprocessed files while use `sic` as their short_name. If the recipe requested `siconc`, the preprocessed files will be identical except that they will use the short_name `siconc` instead.

7.4.5 Diagnostic and variable specific datasets

The `additional_datasets` option can be used to add datasets beyond those listed in the *Datasets* section. This is useful if specific datasets need to be used only by a specific diagnostic or variable, i.e. it can be added both at diagnostic level, where it will apply to all variables in that diagnostic section or at individual variable level. For example, this can be a good way to add observational datasets, which are usually variable-specific.

7.4.6 Running a simple diagnostic

The following example, taken from `recipe_ocean_example.yml`, shows a diagnostic named *diag_map*, which loads the temperature at the ocean surface between the years 2001 and 2003 and then passes it to the `prep_map` preprocessor. The result of this process is then passed to the ocean diagnostic map script, `ocean/diagnostic_maps.py`.

```
diagnostics:

  diag_map:
    description: Global Ocean Surface regridded temperature map
    variables:
      tos: # Temperature at the ocean surface
        preprocessor: prep_map
        start_year: 2001
        end_year: 2003
    scripts:
      Global_Ocean_Surface_regrid_map:
        script: ocean/diagnostic_maps.py
```

7.4.7 Passing arguments to a diagnostic script

The diagnostic script section(s) may include custom arguments that can be used by the diagnostic script; these arguments are stored at runtime in a dictionary that is then made available to the diagnostic script via the interface link, independent of the language the diagnostic script is written in. Here is an example of such groups of arguments:

```
scripts:
  autoassess_strato_test_1: &autoassess_strato_test_1_settings
    script: autoassess/autoassess_area_base.py
    title: "Autoassess Stratosphere Diagnostic Metric MPI-MPI"
    area: stratosphere
    control_model: MPI-ESM-LR
    exp_model: MPI-ESM-MR
    obs_models: [ERA-Interim]  # list to hold models that are NOT for metrics but for␣
↪obs operations
    additional_metrics: [ERA-Interim, inmcm4]  # list to hold additional datasets for␣
↪metrics
```

In this example, apart from specifying the diagnostic script `script: autoassess/autoassess_area_base.py`, we pass a suite of parameters to be used by the script (`area`, `control_model` etc). These parameters are stored in key-value pairs in the diagnostic configuration file, an interface file that can be used by importing the `run_diagnostic` utility:

```
from esmvaltool.diag_scripts.shared import run_diagnostic

# write the diagnostic code here e.g.
def run_some_diagnostic(my_area, my_control_model, my_exp_model):
    """Diagnostic to be run."""
    if my_area == 'stratosphere':
        diag = my_control_model / my_exp_model
        return diag

def main(cfg):
```

(continues on next page)

(continued from previous page)

```
    """Main diagnostic run function."""
    my_area = cfg['area']
    my_control_model = cfg['control_model']
    my_exp_model = cfg['exp_model']
    run_some_diagnostic(my_area, my_control_model, my_exp_model)

if __name__ == '__main__':

    with run_diagnostic() as config:
        main(config)
```

This way a lot of the optional arguments necessary to a diagnostic are at the user's control via the recipe.

7.4.8 Running your own diagnostic

If the user wants to test a newly-developed `my_first_diagnostic.py` which is not yet part of the ESMValTool diagnostics library, he/she do it by passing the absolute path to the diagnostic:

```
diagnostics:

  myFirstDiag:
    description: John Doe wrote a funny diagnostic
    variables:
      tos: # Temperature at the ocean surface
        preprocessor: prep_map
        start_year: 2001
        end_year: 2003
    scripts:
      JoeDiagFunny:
        script: /home/users/john_doe/esmvaltool_testing/my_first_diagnostic.py
```

This way the user may test a new diagnostic thoroughly before committing to the GitHub repository and including it in the ESMValTool diagnostics library.

7.4.9 Re-using parameters from one `script` to another

Due to `yaml` features it is possible to recycle entire diagnostics sections for use with other diagnostics. Here is an example:

```
scripts:
  cycle: &cycle_settings
    script: perfmetrics/main.ncl
    plot_type: cycle
    time_avg: monthlyclim
  grading: &grading_settings
    <<: *cycle_settings
    plot_type: cycle_latlon
    calc_grading: true
    normalization: [centered_median, none]
```

In this example the hook `&cycle_settings` can be used to pass the `cycle:` parameters to `grading:` via the shortcut `<<: *cycle_settings`.

CHAPTER

EIGHT

PREPROCESSOR

In this section, each of the preprocessor modules is described, roughly following the default order in which preprocessor functions are applied:

- *Variable derivation*
- *CMORization and dataset-specific fixes*
- *Fx variables as cell measures or ancillary variables*
- *Vertical interpolation*
- *Weighting*
- *Land-sea masking*
- *Horizontal regridding*
- *Missing values masks*
- *Multi-model statistics*
- *Time manipulation*
- *Area manipulation*
- *Volume manipulation*
- *Cycles*
- *Trend*
- *Detrend*
- *Unit conversion*
- *Other*

See *Preprocessor functions* for implementation details and the exact default order.

8.1 Overview

The ESMValTool preprocessor can be used to perform a broad range of operations on the input data before diagnostics or metrics are applied. The preprocessor performs these operations in a centralized, documented and efficient way, thus reducing the data processing load on the diagnostics side. For an overview of the preprocessor structure see the *Recipe section: preprocessors*.

Each of the preprocessor operations is written in a dedicated python module and all of them receive and return an instance of `iris.cube.Cube`, working sequentially on the data with no interactions between them. The order in which the preprocessor operations is applied is set by default to minimize the loss of information due to, for example,

temporal and spatial subsetting or multi-model averaging. Nevertheless, the user is free to change such order to address specific scientific requirements, but keeping in mind that some operations must be necessarily performed in a specific order. This is the case, for instance, for multi-model statistics, which required the model to be on a common grid and therefore has to be called after the regridding module.

8.2 Variable derivation

The variable derivation module allows to derive variables which are not in the CMIP standard data request using standard variables as input. The typical use case of this operation is the evaluation of a variable which is only available in an observational dataset but not in the models. In this case a derivation function is provided by the ESMValTool in order to calculate the variable and perform the comparison. For example, several observational datasets deliver total column ozone as observed variable (*toz*), but CMIP models only provide the ozone 3D field. In this case, a derivation function is provided to vertically integrate the ozone and obtain total column ozone for direct comparison with the observations.

To contribute a new derived variable, it is also necessary to define a name for it and to provide the corresponding CMOR table. This is to guarantee the proper metadata definition is attached to the derived data. Such custom CMOR tables are collected as part of the ESMValCore package. By default, the variable derivation will be applied only if the variable is not already available in the input data, but the derivation can be forced by setting the appropriate flag.

```
variables:
  toz:
    derive: true
    force_derivation: false
```

The required arguments for this module are two boolean switches:

- `derive`: activate variable derivation
- `force_derivation`: force variable derivation even if the variable is directly available in the input data.

See also *esmvalcore.preprocessor.derive()*. To get an overview on derivation scripts and how to implement new ones, please go to *Deriving a variable*.

8.3 CMORization and dataset-specific fixes

8.3.1 Data checking

Data preprocessed by ESMValTool is automatically checked against its cmor definition. To reduce the impact of this check while maintaining it as reliable as possible, it is split in two parts: one will check the metadata and will be done just after loading and concatenating the data and the other one will check the data itself and will be applied after all extracting operations are applied to reduce the amount of data to process.

Checks include, but are not limited to:

- Requested coordinates are present and comply with their definition.
- Correctness of variable names, units and other metadata.
- Compliance with the valid minimum and maximum values allowed if defined.

The most relevant (i.e. a missing coordinate) will raise an error while others (i.e an incorrect long name) will be reported as a warning.

Some of those issues will be fixed automatically by the tool, including the following:

- Incorrect standard or long names.
- Incorrect units, if they can be converted to the correct ones.
- Direction of coordinates.
- Automatic clipping of longitude to 0 - 360 interval.

8.3.2 Dataset specific fixes

Sometimes, the checker will detect errors that it can not fix by itself. ESMValTool deals with those issues by applying specific fixes for those datasets that require them. Fixes are applied at three different preprocessor steps:

- fix_file: apply fixes directly to a copy of the file. Copying the files is costly, so only errors that prevent Iris to load the file are fixed here. See *esmvalcore.preprocessor.fix_file()*
- fix_metadata: metadata fixes are done just before concatenating the cubes loaded from different files in the final one. Automatic metadata fixes are also applied at this step. See *esmvalcore.preprocessor.fix_metadata()*
- fix_data: data fixes are applied before starting any operation that will alter the data itself. Automatic data fixes are also applied at this step. See *esmvalcore.preprocessor.fix_data()*

To get an overview on data fixes and how to implement new ones, please go to *Fixing data*.

8.4 Fx variables as cell measures or ancillary variables

The following preprocessor may require the use of `fx_variables` to be able to perform the computations:

- `area_statistics`
- `mask_landsea`
- `mask_landseaice`
- `volume_statistics`
- `weighting_landsea_fraction`

The preprocessor step `add_fx_variables` loads the required `fx_variables`, checks them against CMOR standards and adds them either as `cell_measure` or `ancillary_variable` inside the cube data. This ensures that the defined preprocessor chain is applied to both `variables` and `fx_variables`.

Note that when calling steps that require `fx_variables` inside diagnostic scripts, the variables are expected to contain the required `cell_measures` or `ancillary_variables`. If missing, they can be added using the following functions:

```
  from esmvalcore.preprocessor import (add_cell_measure, add_ancillary_variable)

  cube_with_area_measure = add_cell_measure(cube, area_cube, 'area')

  cube_with_volume_measure = add_cell_measure(cube, volume_cube, 'volume)

  cube_with_ancillary_sftlf = add_ancillary_variable(cube, sftlf_cube)

  cube_with_ancillary_sftgif = add_ancillary_variable(cube, sftgif_cube)

Details on the arguments needed for each step can be found in the following sections.
```

8.5 Vertical interpolation

Vertical level selection is an important aspect of data preprocessing since it allows the scientist to perform a number of metrics specific to certain levels (whether it be air pressure or depth, e.g. the Quasi-Biennial-Oscillation (QBO) u30 is computed at 30 hPa). Dataset native vertical grids may not come with the desired set of levels, so an interpolation operation will be needed to regrid the data vertically. ESMValTool can perform this vertical interpolation via the `extract_levels` preprocessor. Level extraction may be done in a number of ways.

Level extraction can be done at specific values passed to `extract_levels` as `levels:` with its value a list of levels (note that the units are CMOR-standard, Pascals (Pa)):

```
preprocessors:
  preproc_select_levels_from_list:
    extract_levels:
      levels: [100000., 50000., 3000., 1000.]
      scheme: linear
```

It is also possible to extract the CMIP-specific, CMOR levels as they appear in the CMOR table, e.g. `plev10` or `plev17` or `plev19` etc:

```
preprocessors:
  preproc_select_levels_from_cmip_table:
    extract_levels:
      levels: {cmor_table: CMIP6, coordinate: plev10}
      scheme: nearest
```

Of good use is also the level extraction with values specific to a certain dataset, without the user actually polling the dataset of interest to find out the specific levels: e.g. in the example below we offer two alternatives to extract the levels and vertically regrid onto the vertical levels of `ERA-Interim`:

```
preprocessors:
  preproc_select_levels_from_dataset:
    extract_levels:
      levels: ERA-Interim
      # This also works, but allows specifying the pressure coordinate name
      # levels: {dataset: ERA-Interim, coordinate: air_pressure}
      scheme: linear_horizontal_extrapolate_vertical
```

By default, vertical interpolation is performed in the dimension coordinate of the z axis. If you want to explicitly declare the z axis coordinate to use (for example, `air_pressure`' in variables that are provided in model levels and not pressure levels) you can override that automatic choice by providing the name of the desired coordinate:

```
preprocessors:
  preproc_select_levels_from_dataset:
    extract_levels:
      levels: ERA-Interim
      scheme: linear_horizontal_extrapolate_vertical
      coordinate: air_pressure
```

If `coordinate` is specified, pressure levels (if present) can be converted to height levels and vice versa using the US standard atmosphere. E.g. `coordinate = altitude` will convert existing pressure levels (air_pressure) to height levels (altitude); `coordinate = air_pressure` will convert existing height levels (altitude) to pressure levels (air_pressure).

- See also *esmvalcore.preprocessor.extract_levels()*.

- See also `esmvalcore.preprocessor.get_cmor_levels()`.

Note: For both vertical and horizontal regridding one can control the extrapolation mode when defining the interpolation scheme. Controlling the extrapolation mode allows us to avoid situations where extrapolating values makes little physical sense (e.g. extrapolating beyond the last data point). The extrapolation mode is controlled by the *extrapolation_mode* keyword. For the available interpolation schemes available in Iris, the extrapolation_mode keyword must be one of:

- `extrapolate`: the extrapolation points will be calculated by extending the gradient of the closest two points;
- `error`: a `ValueError` exception will be raised, notifying an attempt to extrapolate;
- `nan`: the extrapolation points will be be set to NaN;
- `mask`: the extrapolation points will always be masked, even if the source data is not a `MaskedArray`; or
- `nanmask`: if the source data is a MaskedArray the extrapolation points will be masked, otherwise they will be set to NaN.

8.6 Weighting

8.6.1 Land/sea fraction weighting

This preprocessor allows weighting of data by land or sea fractions. In other words, this function multiplies the given input field by a fraction in the range 0-1 to account for the fact that not all grid points are completely land- or sea-covered.

The application of this preprocessor is very important for most carbon cycle variables (and other land surface outputs), which are e.g. reported in units of $kgC\ m^{-2}$. Here, the surface unit actually refers to 'square meter of land/sea' and NOT 'square meter of gridbox'. In order to integrate these globally or regionally one has to weight by both the surface quantity and the land/sea fraction.

For example, to weight an input field with the land fraction, the following preprocessor can be used:

```
preprocessors:
  preproc_weighting:
    weighting_landsea_fraction:
      area_type: land
      exclude: ['CanESM2', 'reference_dataset']
```

Allowed arguments for the keyword `area_type` are `land` (fraction is 1 for grid cells with only land surface, 0 for grid cells with only sea surface and values in between 0 and 1 for coastal regions) and `sea` (1 for sea, 0 for land, in between for coastal regions). The optional argument `exclude` allows to exclude specific datasets from this preprocessor, which is for example useful for climate models which do not offer land/sea fraction files. This arguments also accepts the special dataset specifiers `reference_dataset` and `alternative_dataset`.

Optionally you can specify your own custom fx variable to be used in cases when e.g. a certain experiment is preferred for fx data retrieval:

```
preprocessors:
  preproc_weighting:
    weighting_landsea_fraction:
      area_type: land
      exclude: ['CanESM2', 'reference_dataset']
      fx_variables:
```

(continues on next page)

(continued from previous page)

```
        sftlf:
          exp: piControl
        sftof:
          exp: piControl
```

or alternatively:

```
preprocessors:
  preproc_weighting:
    weighting_landsea_fraction:
      area_type: land
      exclude: ['CanESM2', 'reference_dataset']
      fx_variables: [
        {'short_name': 'sftlf', 'exp': 'piControl'},
        {'short_name': 'sftof', 'exp': 'piControl'}
        ]
```

See also *esmvalcore.preprocessor.weighting_landsea_fraction()*.

8.7 Masking

8.7.1 Introduction to masking

Certain metrics and diagnostics need to be computed and performed on specific domains on the globe. The ESMValTool preprocessor supports filtering the input data on continents, oceans/seas and ice. This is achieved by masking the model data and keeping only the values associated with grid points that correspond to, e.g., land, ocean or ice surfaces, as specified by the user. Where possible, the masking is realized using the standard mask files provided together with the model data as part of the CMIP data request (the so-called fx variable). In the absence of these files, the Natural Earth masks are used: although these are not model-specific, they represent a good approximation since they have a much higher resolution than most of the models and they are regularly updated with changing geographical features.

8.7.2 Land-sea masking

In ESMValTool, land-sea-ice masking can be done in two places: in the preprocessor, to apply a mask on the data before any subsequent preprocessing step and before running the diagnostic, or in the diagnostic scripts themselves. We present both these implementations below.

To mask out a certain domain (e.g., sea) in the preprocessor, `mask_landsea` can be used:

```
preprocessors:
  preproc_mask:
    mask_landsea:
      mask_out: sea
```

and requires only one argument: `mask_out`: either `land` or `sea`.

The preprocessor automatically retrieves the corresponding mask (`fx: stfof` in this case) and applies it so that sea-covered grid cells are set to missing. Conversely, it retrieves the `fx: sftlf` mask when land needs to be masked out, respectively.

Optionally you can specify your own custom fx variable to be used in cases when e.g. a certain experiment is preferred for fx data retrieval. Note that it is possible to specify as many tags for the fx variable as required:

```
preprocessors:
  landmask:
    mask_landsea:
      mask_out: sea
      fx_variables:
        sftlf:
          exp: piControl
        sftof:
          exp: piControl
          ensemble: r2i1p1f1
```

or alternatively:

```
preprocessors:
  landmask:
    mask_landsea:
      mask_out: sea
      fx_variables: [
        {'short_name': 'sftlf', 'exp': 'piControl'},
        {'short_name': 'sftof', 'exp': 'piControl', 'ensemble': 'r2i1p1f1'}
        ]
```

If the corresponding fx file is not found (which is the case for some models and almost all observational datasets), the preprocessor attempts to mask the data using Natural Earth mask files (that are vectorized rasters). As mentioned above, the spatial resolution of the the Natural Earth masks are much higher than any typical global model (10m for land and glaciated areas and 50m for ocean masks).

See also *esmvalcore.preprocessor.mask_landsea()*.

8.7.3 Ice masking

Note that for masking out ice sheets, the preprocessor uses a different function, to ensure that both land and sea or ice can be masked out without losing generality. To mask ice out, `mask_landseaice` can be used:

```
preprocessors:
  preproc_mask:
    mask_landseaice:
      mask_out: ice
```

and requires only one argument: `mask_out`: either `landsea` or `ice`.

As in the case of `mask_landsea`, the preprocessor automatically retrieves the `fx_variables: [sftgif]` mask.

Optionally you can specify your own custom fx variable to be used in cases when e.g. a certain experiment is preferred for fx data retrieval:

```
preprocessors:
  landseaicemask:
    mask_landseaice:
      mask_out: sea
      fx_variables:
        sftgif:
          exp: piControl
```

or alternatively:

```
preprocessors:
  landseaicemask:
    mask_landseaice:
      mask_out: sea
      fx_variables: [{'short_name': 'sftgif', 'exp': 'piControl'}]
```

See also *esmvalcore.preprocessor.mask_landseaice()*.

8.7.4 Glaciated masking

For masking out glaciated areas a Natural Earth shapefile is used. To mask glaciated areas out, `mask_glaciated` can be used:

```
preprocessors:
  preproc_mask:
    mask_glaciated:
      mask_out: glaciated
```

and it requires only one argument: `mask_out`: only `glaciated`.

See also *esmvalcore.preprocessor.mask_landseaice()*.

8.7.5 Missing values masks

Missing (masked) values can be a nuisance especially when dealing with multi-model ensembles and having to compute multi-model statistics; different numbers of missing data from dataset to dataset may introduce biases and artificially assign more weight to the datasets that have less missing data. This is handled in ESMValTool via the missing values masks: two types of such masks are available, one for the multi-model case and another for the single model case.

The multi-model missing values mask (`mask_fillvalues`) is a preprocessor step that usually comes after all the single-model steps (regridding, area selection etc) have been performed; in a nutshell, it combines missing values masks from individual models into a multi-model missing values mask; the individual model masks are built according to common criteria: the user chooses a time window in which missing data points are counted, and if the number of missing data points relative to the number of total data points in a window is less than a chosen fractional threshold, the window is discarded i.e. all the points in the window are masked (set to missing).

```
preprocessors:
  missing_values_preprocessor:
    mask_fillvalues:
      threshold_fraction: 0.95
      min_value: 19.0
      time_window: 10.0
```

In the example above, the fractional threshold for missing data vs. total data is set to 95% and the time window is set to 10.0 (units of the time coordinate units). Optionally, a minimum value threshold can be applied, in this case it is set to 19.0 (in units of the variable units).

See also *esmvalcore.preprocessor.mask_fillvalues()*.

8.7.6 Common mask for multiple models

To create a combined multi-model mask (all the masks from all the analyzed datasets combined into a single mask using a logical OR), the preprocessor `mask_multimodel` can be used. In contrast to `mask_fillvalues`, `mask_multimodel` does not expect that the datasets have a `time` coordinate, but works on datasets with arbitrary (but identical) coordinates. After `mask_multimodel`, all involved datasets have an identical mask.

See also *esmvalcore.preprocessor.mask_multimodel()*.

8.7.7 Minimum, maximum and interval masking

Thresholding on minimum and maximum accepted data values can also be performed: masks are constructed based on the results of thresholding; inside and outside interval thresholding and masking can also be performed. These functions are `mask_above_threshold`, `mask_below_threshold`, `mask_inside_range`, and `mask_outside_range`.

These functions always take a cube as first argument and either `threshold` for threshold masking or the pair `minimum`, `maximum` for interval masking.

See also *esmvalcore.preprocessor.mask_above_threshold()* and related functions.

8.8 Horizontal regridding

Regridding is necessary when various datasets are available on a variety of *lat-lon* grids and they need to be brought together on a common grid (for various statistical operations e.g. multi-model statistics or for e.g. direct inter-comparison or comparison with observational datasets). Regridding is conceptually a very similar process to interpolation (in fact, the regridder engine uses interpolation and extrapolation, with various schemes). The primary difference is that interpolation is based on sample data points, while regridding is based on the horizontal grid of another cube (the reference grid). If the horizontal grids of a cube and its reference grid are sufficiently the same, regridding is automatically and silently skipped for performance reasons.

The underlying regridding mechanism in ESMValTool uses `iris.cube.Cube.regrid` from Iris.

The use of the horizontal regridding functionality is flexible depending on what type of reference grid and what interpolation scheme is preferred. Below we show a few examples.

8.8.1 Regridding on a reference dataset grid

The example below shows how to regrid on the reference dataset `ERA-Interim` (observational data, but just as well CMIP, obs4mips, or ana4mips datasets can be used); in this case the *scheme* is *linear*.

```
preprocessors:
  regrid_preprocessor:
    regrid:
      target_grid: ERA-Interim
      scheme: linear
```

8.8.2 Regridding on an MxN grid specification

The example below shows how to regrid on a reference grid with a cell specification of 2.5x2.5 degrees. This is similar to regridding on reference datasets, but in the previous case the reference dataset grid cell specifications are not necessarily known a priori. Regridding on an MxN cell specification is oftentimes used when operating on localized data.

```
preprocessors:
  regrid_preprocessor:
    regrid:
      target_grid: 2.5x2.5
      scheme: nearest
```

In this case the NearestNeighbour interpolation scheme is used (see below for scheme definitions).

When using a MxN type of grid it is possible to offset the grid cell centrepoints using the *lat_offset* and lon_offset arguments:

- lat_offset: offsets the grid centers of the latitude coordinate w.r.t. the pole by half a grid step;
- lon_offset: offsets the grid centers of the longitude coordinate w.r.t. Greenwich meridian by half a grid step.

```
preprocessors:
  regrid_preprocessor:
    regrid:
      target_grid: 2.5x2.5
      lon_offset: True
      lat_offset: True
      scheme: nearest
```

8.8.3 Regridding to a regional target grid specification

This example shows how to regrid to a regional target grid specification. This is useful if both a regrid and extract_region step are necessary.

```
preprocessors:
  regrid_preprocessor:
    regrid:
      target_grid:
        start_longitude: 40
        end_longitude: 60
        step_longitude: 2
        start_latitude: -10
        end_latitude: 30
        step_latitude: 2
      scheme: nearest
```

This defines a grid ranging from 40° to 60° longitude with 2° steps, and -10° to 30° latitude with 2° steps. If end_longitude or end_latitude do not fall on the grid (e.g., end_longitude: 61), it cuts off at the nearest previous value (e.g. 60).

The longitude coordinates will wrap around the globe if necessary, i.e. start_longitude: 350, end_longitude: 370 is valid input.

The arguments are defined below:

- `start_latitude`: Latitude value of the first grid cell center (start point). The grid includes this value.
- `end_latitude`: Latitude value of the last grid cell center (end point). The grid includes this value only if it falls on a grid point. Otherwise, it cuts off at the previous value.
- `step_latitude`: Latitude distance between the centers of two neighbouring cells.
- `start_longitude`: Latitude value of the first grid cell center (start point). The grid includes this value.
- `end_longitude`: Longitude value of the last grid cell center (end point). The grid includes this value only if it falls on a grid point. Otherwise, it cuts off at the previous value.
- `step_longitude`: Longitude distance between the centers of two neighbouring cells.

8.8.4 Regridding (interpolation, extrapolation) schemes

The schemes used for the interpolation and extrapolation operations needed by the horizontal regridding functionality directly map to their corresponding implementations in Iris:

- `linear`: `Linear(extrapolation_mode='mask')`, see `iris.analysis.Linear`.
- `linear_extrapolate`: `Linear(extrapolation_mode='extrapolate')`, see `iris.analysis.Linear`.
- `nearest`: `Nearest(extrapolation_mode='mask')`, see `iris.analysis.Nearest`.
- `area_weighted`: `AreaWeighted()`, see `iris.analysis.AreaWeighted`.
- `unstructured_nearest`: `UnstructuredNearest()`, see `iris.analysis.UnstructuredNearest`.

See also *esmvalcore.preprocessor.regrid()*

Note: For both vertical and horizontal regridding one can control the extrapolation mode when defining the interpolation scheme. Controlling the extrapolation mode allows us to avoid situations where extrapolating values makes little physical sense (e.g. extrapolating beyond the last data point). The extrapolation mode is controlled by the *extrapolation_mode* keyword. For the available interpolation schemes available in Iris, the extrapolation_mode keyword must be one of:

- `extrapolate` – the extrapolation points will be calculated by extending the gradient of the closest two points;
- `error` – a `ValueError` exception will be raised, notifying an attempt to extrapolate;
- `nan` – the extrapolation points will be be set to NaN;
- `mask` – the extrapolation points will always be masked, even if the source data is not a `MaskedArray`; or
- `nanmask` – if the source data is a MaskedArray the extrapolation points will be masked, otherwise they will be set to NaN.

Note: The regridding mechanism is (at the moment) done with fully realized data in memory, so depending on how fine the target grid is, it may use a rather large amount of memory. Empirically target grids of up to `0.5x0.5` degrees should not produce any memory-related issues, but be advised that for resolutions of `< 0.5` degrees the regridding becomes very slow and will use a lot of memory.

8.9 Multi-model statistics

Computing multi-model statistics is an integral part of model analysis and evaluation: individual models display a variety of biases depending on model set-up, initial conditions, forcings and implementation; comparing model data to observational data, these biases have a significantly lower statistical impact when using a multi-model ensemble. ESMValTool has the capability of computing a number of multi-model statistical measures: using the preprocessor module `multi_model_statistics` will enable the user to ask for either a multi-model `mean`, `median`, `max`, `min`, `std`, and / or `pXX.YY` with a set of argument parameters passed to `multi_model_statistics`. Percentiles can be specified like `p1.5` or `p95`. The decimal point will be replaced by a dash in the output file.

Restrictive computation is also available by excluding any set of models that the user will not want to include in the statistics (by setting `exclude: [excluded models list]` argument). The implementation has a few restrictions that apply to the input data: model datasets must have consistent shapes, apart from the time dimension; and cubes with more than four dimensions (time, vertical axis, two horizontal axes) are not supported.

Input datasets may have different time coordinates. Statistics can be computed across overlapping times only (`span: overlap`) or across the full time span of the combined models (`span: full`). The preprocessor sets a common time coordinate on all datasets. As the number of days in a year may vary between calendars, (sub-)daily data with different calendars are not supported.

Input datasets may have different time coordinates. The multi-model statistics preprocessor sets a common time coordinate on all datasets. As the number of days in a year may vary between calendars, (sub-)daily data are not supported.

```yaml
preprocessors:
  multi_model_preprocessor:
    multi_model_statistics:
      span: overlap
      statistics: [mean, median]
      exclude: [NCEP]
```

see also *esmvalcore.preprocessor.multi_model_statistics()*.

When calling the module inside diagnostic scripts, the input must be given as a list of cubes. The output will be saved in a dictionary where each entry contains the resulting cube with the requested statistic operations.

```
from esmvalcore.preprocessor import multi_model_statistics
statistics = multi_model_statistics([cube1,...,cubeN], 'overlap', ['mean', 'median'])
mean_cube = statistics['mean']
median_cube = statistics['median']
```

Note: The multi-model array operations can be rather memory-intensive (since they are not performed lazily as yet). The Section on *Information on maximum memory required* details the memory intake for different run scenarios, but as a thumb rule, for the multi-model preprocessor, the expected maximum memory intake could be approximated as the number of datasets multiplied by the average size in memory for one dataset.

8.10 Time manipulation

The `_time.py` module contains the following preprocessor functions:

- *extract_time*: Extract a time range from a cube.
- *extract_season*: Extract only the times that occur within a specific season.
- *extract_month*: Extract only the times that occur within a specific month.
- *hourly_statistics*: Compute intra-day statistics
- *daily_statistics*: Compute statistics for each day
- *monthly_statistics*: Compute statistics for each month
- *seasonal_statistics*: Compute statistics for each season
- *annual_statistics*: Compute statistics for each year
- *decadal_statistics*: Compute statistics for each decade
- *climate_statistics*: Compute statistics for the full period
- *resample_time*: Resample data
- *resample_hours*: Convert between N-hourly frequencies by resampling
- *anomalies*: Compute (standardized) anomalies
- *regrid_time*: Aligns the time axis of each dataset to have common time points and calendars.
- *timeseries_filter*: Allows application of a filter to the time-series data.

Statistics functions are applied by default in the order they appear in the list. For example, the following example applied to hourly data will retrieve the minimum values for the full period (by season) of the monthly mean of the daily maximum of any given variable.

```
daily_statistics:
  operator: max

monthly_statistics:
  operator: mean

climate_statistics:
  operator: min
  period: season
```

8.10.1 extract_time

This function subsets a dataset between two points in times. It removes all times in the dataset before the first time and after the last time point. The required arguments are relatively self explanatory:

- `start_year`
- `start_month`
- `start_day`
- `end_year`
- `end_month`

- end_day

These start and end points are set using the datasets native calendar. All six arguments should be given as integers - the named month string will not be accepted.

See also *esmvalcore.preprocessor.extract_time()*.

8.10.2 extract_season

Extract only the times that occur within a specific season.

This function only has one argument: `season`. This is the named season to extract, i.e. DJF, MAM, JJA, SON, but also all other sequentially correct combinations, e.g. JJAS.

Note that this function does not change the time resolution. If your original data is in monthly time resolution, then this function will return three monthly datapoints per year.

If you want the seasonal average, then this function needs to be combined with the seasonal_mean function, below.

See also *esmvalcore.preprocessor.extract_season()*.

8.10.3 extract_month

The function extracts the times that occur within a specific month. This function only has one argument: `month`. This value should be an integer between 1 and 12 as the named month string will not be accepted.

See also *esmvalcore.preprocessor.extract_month()*.

8.10.4 hourly_statistics

This function produces statistics at a x-hourly frequency.

Parameters:

- every_n_hours: frequency to use to compute the statistics. Must be a divisor of 24.
- operator: operation to apply. Accepted values are 'mean', 'median', 'std_dev', 'min', 'max' and 'sum'. Default is 'mean'

See also *esmvalcore.preprocessor.daily_statistics()*.

8.10.5 daily_statistics

This function produces statistics for each day in the dataset.

Parameters:

- operator: operation to apply. Accepted values are 'mean', 'median', 'std_dev', 'min', 'max', 'sum' and 'rms'. Default is 'mean'

See also *esmvalcore.preprocessor.daily_statistics()*.

8.10.6 monthly_statistics

This function produces statistics for each month in the dataset.

Parameters:

- operator: operation to apply. Accepted values are 'mean', 'median', 'std_dev', 'min', 'max', 'sum' and 'rms'. Default is 'mean'

See also *esmvalcore.preprocessor.monthly_statistics()*.

8.10.7 seasonal_statistics

This function produces statistics for each season (default: "(DJF, MAM, JJA, SON)" or custom seasons e.g. "(JJAS, ONDJFMAM)") in the dataset. Note that this function will not check for missing time points. For instance, if you are looking at the DJF field, but your datasets starts on January 1st, the first DJF field will only contain data from January and February.

We recommend using the extract_time to start the dataset from the following December and remove such biased initial datapoints.

Parameters:

- operator: operation to apply. Accepted values are 'mean', 'median', 'std_dev', 'min', 'max', 'sum' and 'rms'. Default is 'mean'
- seasons: seasons to build statistics. Default is '(DJF, MAM, JJA, SON)'

See also `esmvalcore.preprocessor.seasonal_mean()`.

8.10.8 annual_statistics

This function produces statistics for each year.

Parameters:

- operator: operation to apply. Accepted values are 'mean', 'median', 'std_dev', 'min', 'max', 'sum' and 'rms'. Default is 'mean'

See also *esmvalcore.preprocessor.annual_statistics()*.

8.10.9 decadal_statistics

This function produces statistics for each decade.

Parameters:

- operator: operation to apply. Accepted values are 'mean', 'median', 'std_dev', 'min', 'max', 'sum' and 'rms'. Default is 'mean'

See also *esmvalcore.preprocessor.decadal_statistics()*.

8.10.10 climate_statistics

This function produces statistics for the whole dataset. It can produce scalars (if the full period is chosen) or daily, monthly or seasonal statics.

Parameters:

- operator: operation to apply. Accepted values are 'mean', 'median', 'std_dev', 'min', 'max', 'sum' and 'rms'. Default is 'mean'
- period: define the granularity of the statistics: get values for the full period, for each month or day of year. Available periods: 'full', 'season', 'seasonal', 'monthly', 'month', 'mon', 'daily', 'day'. Default is 'full'
- seasons: if period 'seasonal' or 'season' allows to set custom seasons. Default is '(DJF, MAM, JJA, SON)'

Examples:

- Monthly climatology:

```
climate_statistics:
    operator: mean
    period: month
```

- Daily maximum for the full period:

```
climate_statistics:
  operator: max
  period: day
```

- Minimum value in the period:

```
climate_statistics:
  operator: min
  period: full
```

See also *esmvalcore.preprocessor.climate_statistics()*.

8.10.11 resample_time

This function changes the frequency of the data in the cube by extracting the timesteps that meet the criteria. It is important to note that it is mainly meant to be used with instantaneous data.

Parameters:

- month: Extract only timesteps from the given month or do nothing if None. Default is *None*
- day: Extract only timesteps from the given day of month or do nothing if None. Default is *None*
- hour: Extract only timesteps from the given hour or do nothing if None. Default is *None*

Examples:

- Hourly data to daily:

```
resample_time:
  hour: 12
```

- Hourly data to monthly:

```
resample_time:
  hour: 12
  day: 15
```

- Daily data to monthly:

```
resample_time:
  day: 15
```

See also *esmvalcore.preprocessor.resample_time()*.

resample_hours:

8.10.12 resample_hours

This function changes the frequency of the data in the cube by extracting the timesteps that belongs to the desired frequency. It is important to note that it is mainly mean to be used with instantaneous data

Parameters:

- interval: New frequency of the data. Must be a divisor of 24
- offset: First desired hour. Default 0. Must be lower than the interval

Examples:

- Convert to 12-hourly, by getting timesteps at 0:00 and 12:00:

```
resample_hours:
  hours: 12
```

- Convert to 12-hourly, by getting timesteps at 6:00 and 18:00:

```
resample_hours:
  hours: 12
  offset: 6
```

See also *esmvalcore.preprocessor.resample_hours()*.

8.10.13 anomalies

This function computes the anomalies for the whole dataset. It can compute anomalies from the full, seasonal, monthly and daily climatologies. Optionally standardized anomalies can be calculated.

Parameters:

- period: define the granularity of the climatology to use: full period, seasonal, monthly or daily. Available periods: 'full', 'season', 'seasonal', 'monthly', 'month', 'mon', 'daily', 'day'. Default is 'full'
- reference: Time slice to use as the reference to compute the climatology on. Can be 'null' to use the full cube or a dictionary with the parameters from *extract_time*. Default is null
- standardize: if true calculate standardized anomalies (default: false)
- seasons: if period 'seasonal' or 'season' allows to set custom seasons. Default is '(DJF, MAM, JJA, SON)'

Examples:

- Anomalies from the full period climatology:

```
anomalies:
```

- Anomalies from the full period monthly climatology:

```
anomalies:
  period: month
```

- Standardized anomalies from the full period climatology:

```
anomalies:
  standardized: true
```

- Standardized Anomalies from the 1979-2000 monthly climatology:

```
anomalies:
  period: month
  reference:
    start_year: 1979
    start_month: 1
    start_day: 1
    end_year: 2000
    end_month: 12
    end_day: 31
  standardize: true
```

See also *esmvalcore.preprocessor.anomalies()*.

8.10.14 regrid_time

This function aligns the time points of each component dataset so that the Iris cubes from different datasets can be subtracted. The operation makes the datasets time points common; it also resets the time bounds and auxiliary coordinates to reflect the artificially shifted time points. Current implementation for monthly and daily data; the `frequency` is set automatically from the variable CMOR table unless a custom `frequency` is set manually by the user in recipe.

See also *esmvalcore.preprocessor.regrid_time()*.

8.10.15 timeseries_filter

This function allows the user to apply a filter to the timeseries data. This filter may be of the user's choice (currently only the `low-pass` Lanczos filter is implemented); the implementation is inspired by this iris example and uses aggregation via `iris.cube.Cube.rolling_window`.

Parameters:

- window: the length of the filter window (in units of cube time coordinate).
- span: period (number of months/days, depending on data frequency) on which weights should be computed e.g. for 2-yearly: span = 24 (2 x 12 months). Make sure span has the same units as the data cube time coordinate.
- filter_type: the type of filter to be applied; default 'lowpass'. Available types: 'lowpass'.
- filter_stats: the type of statistic to aggregate on the rolling window; default 'sum'. Available operators: 'mean', 'median', 'std_dev', 'sum', 'min', 'max', 'rms'.

Examples:

- Lowpass filter with a monthly mean as operator:

```
timeseries_filter:
    window: 3  # 3-monthly filter window
    span: 12   # weights computed on the first year
    filter_type: lowpass  # low-pass filter
    filter_stats: mean    # 3-monthly mean lowpass filter
```

See also *esmvalcore.preprocessor.timeseries_filter()*.

8.11 Area manipulation

The area manipulation module contains the following preprocessor functions:

- *extract_region*: Extract a region from a cube based on `lat/lon` corners.
- *extract_named_regions*: Extract a specific region from in the region coordinate.
- *extract_shape*: Extract a region defined by a shapefile.
- *extract_point*: Extract a single point (with interpolation)
- *zonal_statistics*: Compute zonal statistics.
- *meridional_statistics*: Compute meridional statistics.
- *area_statistics*: Compute area statistics.

8.11.1 extract_region

This function returns a subset of the data on the rectangular region requested. The boundaries of the region are provided as latitude and longitude coordinates in the arguments:

- `start_longitude`
- `end_longitude`
- `start_latitude`
- `end_latitude`

Note that this function can only be used to extract a rectangular region. Use `extract_shape` to extract any other shaped region from a shapefile.

If the grid is irregular, the returned region retains the original coordinates, but is cropped to a rectangular bounding box defined by the start/end coordinates. The deselected area inside the region is masked.

See also *esmvalcore.preprocessor.extract_region()*.

8.11.2 extract_named_regions

This function extracts a specific named region from the data. This function takes the following argument: `regions` which is either a string or a list of strings of named regions. Note that the dataset must have a `region` coordinate which includes a list of strings as values. This function then matches the named regions against the requested string.

See also *esmvalcore.preprocessor.extract_named_regions()*.

8.11.3 extract_shape

Extract a shape or a representative point for this shape from the data.

Parameters:

- `shapefile`: path to the shapefile containing the geometry of the region to be extracted. If the file contains multiple shapes behaviour depends on the decomposed parameter. This path can be relative to `auxiliary_data_dir` defined in the *User configuration file*.
- **`method`: the method to select the region, selecting either all points** contained by the shape or a single representative point. Choose either 'contains' or 'representative'. If not a single grid point is contained in the shape, a representative point will be selected.
- **`crop`: by default *extract_region* will be used to crop the data to a** minimal rectangular region containing the shape. Set to `false` to only mask data outside the shape. Data on irregular grids will not be cropped.
- `decomposed`: by default `false`, in this case the union of all the regions in the shape file is masked out. If `true`, the regions in the shapefiles are masked out separately, generating an auxiliary dimension for the cube for this.
- `ids`: by default, `[]`, in this case all the shapes in the file will be used. If a list of IDs is provided, only the shapes matching them will be used. The IDs are assigned from the `name` or `id` attributes (in that order of priority) if present in the file or from the reading order if otherwise not present. So, for example, if a file has both `` `name `` and `id` attributes, the ids will be assigned from `name`. If the file only has the `id` attribute, it will be taken from it and if no `name` nor `id` attributes are present, an integer id starting from 1 will be assigned automatically when reading the shapes. We discourage to rely on this last behaviour as we can not assure that the reading order will be the same in different platforms, so we encourage you to modify the file to add a proper id attribute. If the file has an id attribute with a name that is not supported, please open an issue so we can add support for it.

Examples:

- Extract the shape of the river Elbe from a shapefile:

```
extract_shape:
  shapefile: Elbe.shp
  method: contains
```

- Extract the shape of several countries:

```
extract_shape:
shapefile: NaturalEarth/Countries/ne_110m_admin_0_countries.shp
decomposed: True
method: contains
ids:
  - Spain
  - France
```

(continues on next page)

(continued from previous page)

```
  - Italy
  - United Kingdom
  - Taiwan
```

See also *esmvalcore.preprocessor.extract_shape()*.

8.11.4 extract_point

Extract a single point from the data. This is done using either nearest or linear interpolation.

Returns a cube with the extracted point(s), and with adjusted latitude and longitude coordinates (see below).

Multiple points can also be extracted, by supplying an array of latitude and/or longitude coordinates. The resulting point cube will match the respective latitude and longitude coordinate to those of the input coordinates. If the input coordinate is a scalar, the dimension will be missing in the output cube (that is, it will be a scalar).

Parameters:

- `cube`: the input dataset cube.
- `latitude`, `longitude`: coordinates (as floating point values) of the point to be extracted. Either (or both) can also be an array of floating point values.
- `scheme`: interpolation scheme: either `'linear'` or `'nearest'`. There is no default.

8.11.5 zonal_statistics

The function calculates the zonal statistics by applying an operator along the longitude coordinate. This function takes one argument:

- `operator`: Which operation to apply: mean, std_dev, median, min, max, sum or rms.

See also `esmvalcore.preprocessor.zonal_means()`.

8.11.6 meridional_statistics

The function calculates the meridional statistics by applying an operator along the latitude coordinate. This function takes one argument:

- `operator`: Which operation to apply: mean, std_dev, median, min, max, sum or rms.

See also `esmvalcore.preprocessor.meridional_means()`.

8.11.7 area_statistics

This function calculates the average value over a region - weighted by the cell areas of the region. This function takes the argument, `operator`: the name of the operation to apply.

This function can be used to apply several different operations in the horizontal plane: mean, standard deviation, median, variance, minimum, maximum and root mean square.

Note that this function is applied over the entire dataset. If only a specific region, depth layer or time period is required, then those regions need to be removed using other preprocessor operations in advance.

The `fx_variables` argument specifies the fx variables that the user wishes to input to the function; the user may specify it calling the variables e.g.

```
fx_variables:
  areacello:
  volcello:
```

or calling the variables and adding specific variable parameters (the key-value pair may be as specific as a CMOR variable can permit):

```
fx_variables:
  areacello:
    mip: Omon
  volcello:
    mip: fx
```

Alternatively, the `fx_variables` argument can also be specified as a list:

```
fx_variables: ['areacello', 'volcello']
```

or as a list of dictionaries:

```
fx_variables: [{'short_name': 'areacello', 'mip': 'Omon'}, {'short_name': 'volcello',
↪'mip': 'fx'}]
```

The recipe parser will automatically find the data files that are associated with these variables and pass them to the function for loading and processing.

See also *esmvalcore.preprocessor.area_statistics()*.

8.12 Volume manipulation

The `_volume.py` module contains the following preprocessor functions:

- `extract_volume`: Extract a specific depth range from a cube.
- `volume_statistics`: Calculate the volume-weighted average.
- `depth_integration`: Integrate over the depth dimension.
- `extract_transect`: Extract data along a line of constant latitude or longitude.
- `extract_trajectory`: Extract data along a specified trajectory.

8.12.1 `extract_volume`

Extract a specific range in the z-direction from a cube. This function takes two arguments, a minimum and a maximum (`z_min` and `z_max`, respectively) in the z-direction.

Note that this requires the requested z-coordinate range to be the same sign as the Iris cube. That is, if the cube has z-coordinate as negative, then `z_min` and `z_max` need to be negative numbers.

See also *esmvalcore.preprocessor.extract_volume()*.

8.12.2 volume_statistics

This function calculates the volume-weighted average across three dimensions, but maintains the time dimension.

This function takes the argument: `operator`, which defines the operation to apply over the volume.

No depth coordinate is required as this is determined by Iris. This function works best when the `fx_variables` provide the cell volume.

The `fx_variables` argument specifies the fx variables that the user wishes to input to the function; the user may specify it calling the variables e.g.

```
fx_variables:
  areacello:
  volcello:
```

or calling the variables and adding specific variable parameters (the key-value pair may be as specific as a CMOR variable can permit):

```
fx_variables:
  areacello:
    mip: Omon
  volcello:
    mip: fx
```

Alternatively, the `fx_variables` argument can also be specified as a list:

```
fx_variables: ['areacello', 'volcello']
```

or as a list of dictionaries:

```
fx_variables: [{'short_name': 'areacello', 'mip': 'Omon'}, {'short_name': 'volcello',
↪'mip': 'fx'}]
```

The recipe parser will automatically find the data files that are associated with these variables and pass them to the function for loading and processing.

See also *esmvalcore.preprocessor.volume_statistics()*.

8.12.3 depth_integration

This function integrates over the depth dimension. This function does a weighted sum along the z-coordinate, and removes the z direction of the output cube. This preprocessor takes no arguments.

See also *esmvalcore.preprocessor.depth_integration()*.

8.12.4 extract_transect

This function extracts data along a line of constant latitude or longitude. This function takes two arguments, although only one is strictly required. The two arguments are `latitude` and `longitude`. One of these arguments needs to be set to a float, and the other can then be either ignored or set to a minimum or maximum value.

For example, if we set latitude to 0 N and leave longitude blank, it would produce a cube along the Equator. On the other hand, if we set latitude to 0 and then set longitude to `[40., 100.]` this will produce a transect of the Equator in the Indian Ocean.

See also *esmvalcore.preprocessor.extract_transect()*.

8.12.5 extract_trajectory

This function extract data along a specified trajectory. The three arguments are: `latitudes`, `longitudes` and number of point needed for extrapolation `number_points`.

If two points are provided, the `number_points` argument is used to set a the number of places to extract between the two end points.

If more than two points are provided, then `extract_trajectory` will produce a cube which has extrapolated the data of the cube to those points, and `number_points` is not needed.

Note that this function uses the expensive `interpolate` method from `Iris.analysis.trajectory`, but it may be necessary for irregular grids.

See also *esmvalcore.preprocessor.extract_trajectory()*.

8.13 Cycles

The `_cycles.py` module contains the following preprocessor functions:

- `amplitude`: Extract the peak-to-peak amplitude of a cycle aggregated over specified coordinates.

8.13.1 amplitude

This function extracts the peak-to-peak amplitude (maximum value minus minimum value) of a field aggregated over specified coordinates. Its only argument is `coords`, which can either be a single coordinate (given as `str`) or multiple coordinates (given as `list` of `str`). Usually, these coordinates refer to temporal categorised coordinates `iris.coord_categorisation` like *year*, *month*, *day of year*, etc. For example, to extract the amplitude of the annual cycle for every single year in the data, use `coords: year`; to extract the amplitude of the diurnal cycle for every single day in the data, use `coords: [year, day_of_year]`.

See also *esmvalcore.preprocessor.amplitude()*.

8.14 Trend

The trend module contains the following preprocessor functions:

- `linear_trend`: Calculate linear trend along a specified coordinate.
- `linear_trend_stderr`: Calculate standard error of linear trend along a specified coordinate.

8.14.1 linear_trend

This function calculates the linear trend of a dataset (defined as slope of an ordinary linear regression) along a specified coordinate. The only argument of this preprocessor is `coordinate` (given as `str`; default value is `'time'`).

See also *esmvalcore.preprocessor.linear_trend()*.

8.14.2 linear_trend_stderr

This function calculates the standard error of the linear trend of a dataset (defined as the standard error of the slope in an ordinary linear regression) along a specified coordinate. The only argument of this preprocessor is `coordinate` (given as `str`; default value is `'time'`). Note that the standard error is **not** identical to a confidence interval.

See also *esmvalcore.preprocessor.linear_trend_stderr()*.

8.15 Detrend

ESMValTool also supports detrending along any dimension using the preprocessor function 'detrend'. This function has two parameters:

- `dimension`: dimension to apply detrend on. Default: "time"
- `method`: It can be `linear` or `constant`. Default: `linear`

If method is `linear`, detrend will calculate the linear trend along the selected axis and subtract it to the data. For example, this can be used to remove the linear trend caused by climate change on some variables is selected dimension is time.

If method is `constant`, detrend will compute the mean along that dimension and subtract it from the data

See also *esmvalcore.preprocessor.detrend()*.

8.16 Unit conversion

Converting units is also supported. This is particularly useful in cases where different datasets might have different units, for example when comparing CMIP5 and CMIP6 variables where the units have changed or in case of observational datasets that are delivered in different units.

In these cases, having a unit conversion at the end of the processing will guarantee homogeneous input for the diagnostics.

Note: Conversion is only supported between compatible units! In other words, converting temperature units from `degC` to `Kelvin` works fine, changing precipitation units from a rate based unit to an amount based unit is not supported at the moment.

See also *esmvalcore.preprocessor.convert_units()*.

8.17 Information on maximum memory required

In the most general case, we can set upper limits on the maximum memory the analysis will require:

`Ms = (R + N) x F_eff - F_eff` - when no multi-model analysis is performed;

`Mm = (2R + N) x F_eff - 2F_eff` - when multi-model analysis is performed;

where

- `Ms`: maximum memory for non-multimodel module
- `Mm`: maximum memory for multi-model module

- `R`: computational efficiency of module; *R* is typically 2-3
- `N`: number of datasets
- `F_eff`: average size of data per dataset where `F_eff = e x f x F` where `e` is the factor that describes how lazy the data is (`e = 1` for fully realized data) and `f` describes how much the data was shrunk by the immediately previous module, e.g. time extraction, area selection or level extraction; note that for fix_data `f` relates only to the time extraction, if data is exact in time (no time selection) `f = 1` for fix_data so for cases when we deal with a lot of datasets `R + N \approx N`, data is fully realized, assuming an average size of 1.5GB for 10 years of *3D* netCDF data, `N` datasets will require:

`Ms = 1.5 x (N - 1)` GB

`Mm = 1.5 x (N - 2)` GB

As a rule of thumb, the maximum required memory at a certain time for multi-model analysis could be estimated by multiplying the number of datasets by the average file size of all the datasets; this memory intake is high but also assumes that all data is fully realized in memory; this aspect will gradually change and the amount of realized data will decrease with the increase of `dask` use.

8.18 Other

Miscellaneous functions that do not belong to any of the other categories.

8.18.1 Clip

This function clips data values to a certain minimum, maximum or range. The function takes two arguments:

- `minimum`: Lower bound of range. Default: `None`
- `maximum`: Upper bound of range. Default: `None`

The example below shows how to set all values below zero to zero.

```
preprocessors:
  clip:
    minimum: 0
    maximum: null
```

Part III

Diagnostic script interfaces

In order to communicate with diagnostic scripts, ESMValCore uses YAML files. The YAML files provided by ESMValCore to the diagnostic script tell the diagnostic script the settings that were provided in the recipe and where to find the pre-processed input data. On the other hand, the YAML file provided by the diagnostic script to ESMValCore tells ESMValCore which pre-processed data was used to create what plots. The latter is optional, but needed for recording provenance.

CHAPTER

NINE

PROVENANCE

When ESMValCore (the `esmvaltool` command) runs a recipe, it will first find all data and run the default preprocessor steps plus any additional preprocessing steps defined in the recipe. Next it will run the diagnostic script defined in the recipe and finally it will store provenance information. Provenance information is stored in the W3C PROV XML format and also plotted in an SVG file for human inspection. In addition to provenance information, a caption is also added to the plots.

CHAPTER

TEN

INFORMATION PROVIDED BY ESMVALCORE TO THE DIAGNOSTIC SCRIPT

To provide the diagnostic script with the information it needs to run (e.g. location of input data, various settings), the ESMValCore creates a YAML file called settings.yml and provides the path to this file as the first command line argument to the diagnostic script.

The most interesting settings provided in this file are

```
run_dir:  /path/to/recipe_output/run/diagnostic_name/script_name
work_dir: /path/to/recipe_output/work/diagnostic_name/script_name
plot_dir: /path/to/recipe_output/plots/diagnostic_name/script_name
input_files:
  - /path/to/recipe_output/preproc/diagnostic_name/ta/metadata.yml
  - /path/to/recipe_output/preproc/diagnostic_name/pr/metadata.yml
```

Custom settings in the script section of the recipe will also be made available in this file.

There are three directories defined:

- `run_dir` use this for storing temporary files
- `work_dir` use this for storing NetCDF files containing the data used to make a plot
- `plot_dir` use this for storing plots

Finally `input_files` is a list of YAML files, containing a description of the preprocessed data. Each entry in these YAML files is a path to a preprocessed file in NetCDF format, with a list of various attributes. An example preprocessor metadata.yml file could look like this:

```
? /path/to/recipe_output/preproc/diagnostic_name/pr/CMIP5_GFDL-ESM2G_Amon_historical_
↪r1i1p1_T2Ms_pr_2000-2002.nc
: alias: GFDL-ESM2G
  cmor_table: CMIP5
  dataset: GFDL-ESM2G
  diagnostic: diagnostic_name
  end_year: 2002
  ensemble: r1i1p1
  exp: historical
  filename: /path/to/recipe_output/preproc/diagnostic_name/pr/CMIP5_GFDL-ESM2G_Amon_
↪historical_r1i1p1_T2Ms_pr_2000-2002.nc
  frequency: mon
  institute: [NOAA-GFDL]
  long_name: Precipitation
  mip: Amon
```

(continues on next page)

(continued from previous page)

```
  modeling_realm: [atmos]
  preprocessor: preprocessor_name
  project: CMIP5
  recipe_dataset_index: 1
  reference_dataset: MPI-ESM-LR
  short_name: pr
  standard_name: precipitation_flux
  start_year: 2000
  units: kg m-2 s-1
  variable_group: pr
? /path/to/recipe_output/preproc/diagnostic_name/pr/CMIP5_MPI-ESM-LR_Amon_historical_
↪r1i1p1_T2Ms_pr_2000-2002.nc
: alias: MPI-ESM-LR
  cmor_table: CMIP5
  dataset: MPI-ESM-LR
  diagnostic: diagnostic_name
  end_year: 2002
  ensemble: r1i1p1
  exp: historical
  filename: /path/to/recipe_output/preproc/diagnostic1/pr/CMIP5_MPI-ESM-LR_Amon_
↪historical_r1i1p1_T2Ms_pr_2000-2002.nc
  frequency: mon
  institute: [MPI-M]
  long_name: Precipitation
  mip: Amon
  modeling_realm: [atmos]
  preprocessor: preprocessor_name
  project: CMIP5
  recipe_dataset_index: 2
  reference_dataset: MPI-ESM-LR
  short_name: pr
  standard_name: precipitation_flux
  start_year: 2000
  units: kg m-2 s-1
  variable_group: pr
```

CHAPTER

ELEVEN

INFORMATION PROVIDED BY THE DIAGNOSTIC SCRIPT TO ESMVALCORE

After the diagnostic script has finished running, ESMValCore will try to store provenance information. In order to link the produced files to input data, the diagnostic script needs to store a YAML file called `diagnostic_provenance.yml` in it's `run_dir`.

For output file produced by the diagnostic script, there should be an entry in the `diagnostic_provenance.yml` file. The name of each entry should be the path to the output file. Each file entry should at least contain the following items

- `ancestors` a list of input files used to create the plot
- `caption` a caption text for the plot
- `plot_file` if the diagnostic also created a plot file, e.g. in .png format.

Each file entry can also contain items from the categories defined in the file `esmvaltool/config_references.yml`. The short entries will automatically be replaced by their longer equivalent in the final provenance records. It is possible to add custom provenance information by adding custom items to entries.

An example `diagnostic_provenance.yml` file could look like this

```
? /path/to/recipe_output/work/diagnostic_name/script_name/CMIP5_GFDL-ESM2G_Amon_
↪historical_r1i1p1_T2Ms_pr_2000-2002_mean.nc
: ancestors:[/path/to/recipe_output/preproc/diagnostic_name/pr/CMIP5_GFDL-ESM2G_Amon_
↪historical_r1i1p1_T2Ms_pr_2000-2002.nc]
  authors: [andela_bouwe, righi_mattia]
  caption: Average Precipitation between 2000 and 2002 according to GFDL-ESM2G.
  domains: [global]
  plot_file: /path/to/recipe_output/plots/diagnostic_name/script_name/CMIP5_GFDL ESM2G_
↪Amon_historical_r1i1p1_T2Ms_pr_2000-2002_mean.png
  plot_type: zonal
  references: [acknow_project]
  statistics: [mean]

? /path/to/recipe_output/work/diagnostic_name/script_name/CMIP5_MPI-ESM-LR_Amon_
↪historical_r1i1p1_T2Ms_pr_2000-2002_mean.nc
: ancestors: [/path/to/recipe_output/preproc/diagnostic_name/pr/CMIP5_MPI-ESM-LR_Amon_
↪historical_r1i1p1_T2Ms_pr_2000-2002.nc]
  authors: [andela_bouwe, righi_mattia]
  caption: Average Precipitation between 2000 and 2002 according to MPI-ESM-LR.
  domains: [global]
  plot_file: /path/to/recipe_output/plots/diagnostic_name/script_name/CMIP5_MPI-ESM-LR_
↪Amon_historical_r1i1p1_T2Ms_pr_2000-2002_mean.png
  plot_type: zonal
```

(continues on next page)

(continued from previous page)

```
  references: [acknow_project]
  statistics: [mean]
```

You can check whether your diagnostic script successfully provided the provenance information to the ESMValCore by verifying that

- for each output file in the `work_dir`, a file with the same name, but ending with _provenance.xml is created
- any NetCDF files created by your diagnostic script contain a 'provenance' global attribute
- any PNG plots created by your diagnostic script contain the provenance information in the 'Image History' attribute

Note that this information is included automatically by ESMValCore if the diagnostic script provides the required `diagnostic_provenance.yml` file.

Part IV

Development

To get started developing, have a look at our *contribution guidelines*. This chapter describes how to implement the most commonly contributed new features.

CHAPTER

TWELVE

PREPROCESSOR FUNCTION

Preprocessor functions are located in *esmvalcore.preprocessor*. To add a new preprocessor function, start by finding a likely looking file to add your function to in esmvalcore/preprocessor. Create a new file in that directory if you cannot find a suitable place.

The function should look like this:

```
def example_preprocessor_function(
    cube,
    example_argument,
    example_optional_argument=5,
):
    """Compute an example quantity.

    A more extensive explanation of the computation can be added here. Add
    references to scientific literature if available.

    Parameters
    ----------
    cube: iris.cube.Cube
       Input cube.

    example_argument: str
       Example argument, the value of this argument can be provided in the
       recipe. Describe what valid values are here. In this case, a valid
       argument is the name of a dimension of the input cube.

    example_optional_argument: int, optional
       Another example argument, the value of this argument can optionally
       be provided in the recipe. Describe what valid values are here.

    Returns
    -------
    iris.cube.Cube
      The result of the example computation.
    """

    # Replace this with your own computation
    cube = cube.collapsed(example_argument, iris.analysis.MEAN)

    return cube
```

The above function needs to be imported in the file esmvalcore/preprocessor/__init__.py:

```
from ._example_module import example_preprocessor_function

__all__ = [
...
'example_preprocessor_function',
...
]
```

The location in the `__all__` list above determines the default order in which preprocessor functions are applied, so carefully consider where you put it and ask for advice if needed.

The preprocessor function above can then be used from the *Recipe section: preprocessors* like this:

```
preprocessors:
  example_preprocessor:
    example_preprocessor_function:
      example_argument: median
      example_optional_argument: 6
```

The optional argument (in this example: `example_optional_argument`) can be omitted in the recipe.

12.1 Lazy and real data

Preprocessor functions should support both real and lazy data. This is vital for supporting the large datasets that are typically used with the ESMValCore. If the data of the incoming cube has been realized (i.e. `cube.has_lazy_data()` returns `False` so `cube.core_data()` is a NumPy array), the returned cube should also have realized data. Conversely, if the incoming cube has lazy data (i.e. `cube.has_lazy_data()` returns `True` so `cube.core_data()` is a Dask array), the returned cube should also have lazy data. Note that NumPy functions will often call their Dask equivalent if it exists and if their input array is a Dask array, and vice versa.

Note that preprocessor functions should preferably be small and just call the relevant iris code. Code that is more involved, e.g. lots of work with Numpy and Dask arrays, and more broadly applicable, should be implemented in iris instead.

12.2 Documentation

The documentation in the function docstring will be shown in the *Preprocessor functions* chapter. In addition, you should add documentation on how to use the new preprocessor function from the recipe in doc/recipe/preprocessor.rst so it is shown in the *Preprocessor* chapter. See the introduction to *Documentation* for more information on how to best write documentation.

12.3 Tests

Tests are should be implemented for new or modified preprocessor functions. For an introduction to the topic, see *Tests*.

12.3.1 Unit tests

To add a unit test for the preprocessor function from the example above, create a file called `tests/unit/preprocessor/_example_module/test_example_preprocessor_function.py` and add the following content:

```
"""Test function `esmvalcore.preprocessor.example_preprocessor_function`."""
import cf_units
import dask.array as da
import iris
import numpy as np
import pytest

from esmvalcore.preprocessor import example_preprocessor_function


@pytest.mark.parametrize('lazy', [True, False])
def test_example_preprocessor_function(lazy):
    """Test that the computed result is as expected."""

    # Construct the input cube
    data = np.array([1, 2], dtype=np.float32)
    if lazy:
        data = da.asarray(data, chunks=(1, ))
    cube = iris.cube.Cube(
        data,
        var_name='tas',
        units='K',
    )
    cube.add_dim_coord(
        iris.coords.DimCoord(
            np.array([0.5, 1.5], dtype=np.float64),
            bounds=np.array([[0, 1], [1, 2]], dtype=np.float64),
            standard_name='time',
            units=cf_units.Unit('days since 1950-01-01 00:00:00',
                                calendar='gregorian'),
        ),
        0,
    )

    # Compute the result
    result = example_preprocessor_function(cube, example_argument='time')

    # Check that lazy data is returned if and only if the input is lazy
    assert result.has_lazy_data() is lazy

    # Construct the expected result cube
    expected = iris.cube.Cube(
        np.array(1.5, dtype=np.float32),
```

(continues on next page)

(continued from previous page)

```
        var_name='tas',
        units='K',
    )
    expected.add_aux_coord(
        iris.coords.AuxCoord(
            np.array([1], dtype=np.float64),
            bounds=np.array([[0, 2]], dtype=np.float64),
            standard_name='time',
            units=cf_units.Unit('days since 1950-01-01 00:00:00',
                                calendar='gregorian'),
        ))
    expected.add_cell_method(
        iris.coords.CellMethod(method='mean', coords=('time', )))

    # Compare the result of the computation with the expected result
    print('result:', result)
    print('expected result:', expected)
    assert result == expected
```

In this test we used the decorator pytest.mark.parametrize to test two scenarios, with both lazy and realized data, with a single test.

12.3.2 Sample data tests

The idea of adding *sample data tests* is to check that preprocessor functions work with realistic data. This also provides an easy way to add regression tests, though these should preferably be implemented as unit tests instead, because using the sample data for this purpose is slow. To add a test using the sample data, create a file `tests/sample_data/preprocessor/example_preprocessor_function/test_example_preprocessor_function.py` and add the following content:

```
"""Test function `esmvalcore.preprocessor.example_preprocessor_function`."""
from pathlib import Path

import esmvaltool_sample_data
import iris
import pytest

from esmvalcore.preprocessor import example_preprocessor_function


@pytest.mark.use_sample_data
def test_example_preprocessor_function():
    """Regression test to check that the computed result is as expected."""
    # Load an example input cube
    cube = esmvaltool_sample_data.load_timeseries_cubes(mip_table='Amon')[0]

    # Compute the result
    result = example_preprocessor_function(cube, example_argument='time')

    filename = Path(__file__).with_name('example_preprocessor_function.nc')
    if not filename.exists():
```

(continues on next page)

(continued from previous page)

```
        # Create the file the expected result if it doesn't exist
        iris.save(result, target=str(filename))
        raise FileNotFoundError(
            f'Reference data was missing, wrote new copy to {filename}')

    # Load the expected result cube
    expected = iris.load_cube(str(filename))

    # Compare the result of the computation with the expected result
    print('result:', result)
    print('expected result:', expected)
    assert result == expected
```

This will use a file from the sample data repository as input. The first time you run the test, the computed result will be stored in the file `tests/sample_data/preprocessor/example_preprocessor_function/example_preprocessor_function.nc` Any subsequent runs will re-load the data from file and check that it did not change. Make sure the stored results are small, i.e. smaller than 100 kilobytes, to keep the size of the ESMValCore repository small.

12.4 Using multiple datasets as input

The name of the first argument of the preprocessor function should in almost all cases be `cube`. Only when implementing a preprocessor function that uses all datasets as input, the name of the first argument should be `products`. If you would like to implement this type of preprocessor function, start by having a look at the existing functions, e.g. *esmvalcore.preprocessor.multi_model_statistics()* or *esmvalcore.preprocessor.mask_fillvalues()*.

CHAPTER

THIRTEEN

FIXING DATA

The baseline case for ESMValCore input data is CMOR fully compliant data that is read using Iris' `iris.load_raw()`. ESMValCore also allows for some departures from compliance (see *Customizing checker strictness*). Beyond that situation, some datasets (either model or observations) contain (known) errors that would normally prevent them from being processed. The issues can be in the metadata describing the dataset and/or in the actual data. Typical examples of such errors are missing or wrong attributes (e.g. attribute ''units" says 1e-9 but data are actually in 1e-6), missing or mislabeled coordinates (e.g. ''lev" instead of ''plev" or missing coordinate bounds like ''lat_bnds") or problems with the actual data (e.g. cloud liquid water only instead of sum of liquid + ice as specified by the CMIP data request).

As an extreme case, some data sources simply are not NetCDF files and must go through some other data load function.

The ESMValCore can apply on the fly fixes to such datasets when issues can be fixed automatically. This is implemented for a set of *Natively supported non-CMIP datasets*. The following provides details on how to design such fixes.

Note: CMORizer scripts. Support for many observational and reanalysis datasets is also possible through a priori reformatting by CMORizer scripts in the ESMValTool, which are rather relevant for datasets of small volume

13.1 Fix structure

Fixes are Python classes stored in `esmvalcore/cmor/_fixes/[PROJECT]/[DATASET].py` that derive from `esmvalcore.cmor._fixes.fix.Fix` and are named after the short name of the variable they fix. You can also use the names of `mip` tables (e.g., `Amon`, `Lmon`, `Omon`, etc.) if you want the fix to be applied to all variables of that table in the dataset or `AllVars` if you want the fix to be applied to the whole dataset.

Warning: Be careful to replace any `-` with `_` in your dataset name. We need this replacement to have proper python module names.

The fixes are automatically loaded and applied when the dataset is preprocessed. They are a special type of *preprocessor function*, called by the preprocessor functions *`esmvalcore.preprocessor.fix_file()`*, *`esmvalcore.preprocessor.fix_metadata()`*, and *`esmvalcore.preprocessor.fix_data()`*.

13.2 Fixing a dataset

To illustrate the process of creating a fix we are going to construct a new one from scratch for a fictional dataset. We need to fix a CMIPX model called PERFECT-MODEL that is reporting a missing latitude coordinate for variable tas.

13.2.1 Check the output

Next to the error message, you should see some info about the iris cube: size, coordinates. In our example it looks like this:

```
air_temperature/ (K) (time: 312; altitude: 90; longitude: 180)
    Dimension coordinates:
        time                                    x              -              -
        altitude                                -              x              -
        longitude                               -              -              x
    Auxiliary coordinates:
        day_of_month                            x              -              -
        day_of_year                             x              -              -
        month_number                            x              -              -
        year                                    x              -              -
    Attributes:
        {'cmor_table': 'CMIPX', 'mip': 'Amon', 'short_name': 'tas', 'frequency': 'mon'})
```

So now the mistake is clear: the latitude coordinate is badly named and the fix should just rename it.

13.2.2 Create the fix

We start by creating the module file. In our example the path will be `esmvalcore/cmor/_fixes/CMIPX/PERFECT_MODEL.py`. If it already exists just add the class to the file, there is no limit in the number of fixes we can have in any given file.

Then we have to create the class for the fix deriving from `esmvalcore.cmor._fixes.Fix`

```
"""Fixes for PERFECT-MODEL."""
from esmvalcore.cmor.fix import Fix

class tas(Fix):
     """Fixes for tas variable."""
```

Next we must choose the method to use between the ones offered by the Fix class:

- `fix_file` : should be used only to fix errors that prevent data loading. As a rule of thumb, you should only use it if the execution halts before reaching the checks.
- `fix_metadata` : you want to change something in the cube that is not the data (e.g variable or coordinate names, data units).
- `fix_data`: you need to fix the data. Beware: coordinates data values are part of the metadata.

In our case we need to rename the coordinate `altitude` to `latitude`, so we will implement the `fix_metadata` method:

```
"""Fixes for PERFECT-MODEL."""
from esmvalcore.cmor.fix import Fix

class tas(Fix):
    """Fixes for tas variable."""

    def fix_metadata(self, cubes):
        """
        Fix metadata for tas.

        Fix the name of the latitude coordinate, which is called altitude
        in the original file.
        """
        # Sometimes Iris will interpret the data as multiple cubes.
        # Good CMOR datasets will only show one but we support the
        # multiple cubes case to be able to fix the errors that are
        # leading to that extra cubes.
        # In our case this means that we can safely assume that the
        # tas cube is the first one
        tas_cube = cubes[0]
        latitude = tas_cube.coord('altitude')

        # Fix the names. Latitude values, units and
        latitude.short_name = 'lat'
        latitude.standard_name = 'latitude'
        latitude.long_name = 'latitude'
        return cubes
```

This will fix the error. The next time you run ESMValTool you will find that the error is fixed on the fly and, hopefully, your recipe will run free of errors. The `cubes` argument to the `fix_metadata` method will contain all cubes loaded from a single input file. Some care may need to be taken that the right cube is selected and fixed in case multiple cubes are created. Usually this happens when a coordinate is mistakenly loaded as a cube, because the input data does not follow the CF Conventions.

Sometimes other errors can appear after you fix the first one because they were hidden by it. In our case, the latitude coordinate could have bad units or values outside the valid range for example. Just extend your fix to address those errors.

13.2.3 Finishing

Chances are that you are not the only one that wants to use that dataset and variable. Other users could take advantage of your fixes as soon as possible. Please, create a separated pull request for the fix and submit it.

It will also be very helpful if you just scan a couple of other variables from the same dataset and check if they share this error. In case that you find that it is a general one, you can change the fix name to the corresponding `mip` table name (e.g., `Amon`, `Lmon`, `Omon`, etc.) so it gets executed for all variables in that table in the dataset or to `AllVars` so it gets executed for all variables in the dataset. If you find that this is shared only by a handful of similar vars you can just make the fix for those new vars derive from the one you just created:

```
"""Fixes for PERFECT-MODEL."""
from esmvalcore.cmor.fix import Fix

class tas(Fix):
```

(continues on next page)

(continued from previous page)

```
    """Fixes for tas variable."""

    def fix_metadata(self, cubes):
        """
        Fix metadata for tas.

        Fix the name of the latitude coordinate, which is called altitude
        in the original file.
        """
        # Sometimes Iris will interpret the data as multiple cubes.
        # Good CMOR datasets will only show one but we support the
        # multiple cubes case to be able to fix the errors that are
        # leading to that extra cubes.
        # In our case this means that we can safely assume that the
        # tas cube is the first one
        tas_cube = cubes[0]
        latitude = tas_cube.coord('altitude')

        # Fix the names. Latitude values, units and
        latitude.short_name = 'lat'
        latitude.standard_name = 'latitude'
        latitude.long_name = 'latitude'
        return cubes


class ps(tas):
    """Fixes for ps variable."""
```

13.3 Common errors

The above example covers one of the most common cases: variables / coordinates that have names that do not match the expected. But there are some others that use to appear frequently. This section describes the most common cases.

13.3.1 Bad units declared

It is quite common that a variable declares to be using some units but the data is stored in another. This can be solved by overwriting the units attribute with the actual data units.

```
def fix_metadata(self, cubes):
    cube.units = 'real_units'
```

Detecting this error can be tricky if the units are similar enough. It also has a good chance of going undetected until you notice strange results in your diagnostic.

For the above example, it can be useful to access the variable definition and associated coordinate definitions as provided by the CMOR table. For example:

```
def fix_metadata(self, cubes):
    cube.units = self.vardef.units
```

To learn more about what is available in these definitions, see: *esmvalcore.cmor.table.VariableInfo* and *esmvalcore.cmor.table.CoordinateInfo*.

13.3.2 Coordinates missing

Another common error is to have missing coordinates. Usually it just means that the file does not follow the CF-conventions and Iris can therefore not interpret it.

If this is the case, you should see a warning from the ESMValTool about discarding some cubes in the fix metadata step. Just before that warning you should see the full list of cubes as read by Iris. If that list contains your missing coordinate you can create a fix for this model:

```
def fix_metadata(self, cubes):
    coord_cube = cubes.extract_strict('COORDINATE_NAME')
    # Usually this will correspond to an auxiliary coordinate
    # because the most common error is to forget adding it to the
    # coordinates attribute
    coord = iris.coords.AuxCoord(
        coord_cube.data,
        var_name=coord_cube.var_name,
        standard_name=coord_cube.standard_name,
        long_name=coord_cube.long_name,
        units=coord_cube.units,
    }

    # It may also have bounds as another cube
    coord.bounds = cubes.extract_strict('BOUNDS_NAME').data

    data_cube = cubes.extract_strict('VAR_NAME')
    data_cube.add_aux_coord(coord, DIMENSIONS_INDEX_TUPLE)
    return [data_cube]
```

13.4 Customizing checker strictness

The data checker classifies its issues using four different levels of severity. From highest to lowest:

- `CRITICAL`: issues that most of the time will have severe consequences.
- `ERROR`: issues that usually lead to unexpected errors, but can be safely ignored sometimes.
- `WARNING`: something is not up to the standard but is unlikely to have consequences later.
- `DEBUG`: any info that the checker wants to communicate. Regardless of checker strictness, those will always be reported as debug messages.

Users can have control about which levels of issues are interpreted as errors, and therefore make the checker fail or warnings or debug messages. For this purpose there is an optional command line option *–check-level* that can take a number of values, listed below from the lowest level of strictness to the highest:

- `ignore`: all issues, regardless of severity, will be reported as warnings. Checker will never fail. Use this at your own risk.
- `relaxed`: only CRITICAL issues are treated as errors. We recommend not to rely on this mode, although it can be useful if there are errors preventing the run that you are sure you can manage on the diagnostics or that will not affect you.

- `default`: fail if there are any CRITICAL or ERROR issues (DEFAULT); Provides a good measure of safety.
- `strict`: fail if there are any warnings, this is the highest level of strictness. Mostly useful for checking datasets that you have produced, to be sure that future users will not be distracted by inoffensive warnings.

13.5 Natively supported non-CMIP datasets

Some fixed datasets and native models formats are supported through the `native6` project or through a dedicated project.

13.5.1 Observational Datasets

Put the files containing the data in the directory that you have configured for the `native6` project in your *User configuration file*, in a subdirectory called `Tier{tier}/{dataset}/{version}/{frequency}/{short_name}`. Replace the items in curly braces by the values used in the variable/dataset definition in the *recipe*. Below is a list of datasets currently supported.

ERA5

- Supported variables: `clt`, `evspsbl`, `evspsblpot`, `mrro`, `pr`, `prsn`, `ps`, `psl`, `ptype`, `rls`, `rlds`, `rsds`, `rsdt`, `rss`, `uas`, `vas`, `tas`, `tasmax`, `tasmin`, `tdps`, `ts`, `tsn` (`E1hr`/`Amon`), `orog` (`fx`)
- Tier: 3

MSWEP

- Supported variables: `pr`
- Supported frequencies: `mon`, `day`, `3hr`.
- Tier: 3

For example for monthly data, place the files in the `/Tier3/MSWEP/latestversion/mon/pr` subdirectory of your `native6` project location.

Note: For monthly data (V220), the data must be postfixed with the date, i.e. rename `global_monthly_050deg.nc` to `global_monthly_050deg_197901-201710.nc`

For more info: http://www.gloh2o.org/

13.5.2 Native models

The following models are natively supported through the procedure described above (*Fix structure*) and at *Configuring native models and observation data sets*:

IPSL-CM6

Both output formats (i.e. the `Output` and the `Analyse / Time series` formats) are supported, and should be configured in recipes as e.g.:

```
datasets:
  - {simulation: CM61-LR-hist-03.1950, exp: piControl, freq: Analyse/TS_MO,
     account: p86caub,  status: PROD, dataset: IPSL-CM6, project: IPSLCM,
     root: /thredds/tgcc/store}
  - {simulation: CM61-LR-hist-03.1950, exp: historical, freq: Output/MO,
     account: p86caub,  status: PROD, dataset: IPSL-CM6, project: IPSLCM,
     root: /thredds/tgcc/store}
```

The `Output` format is an example of a case where variables are grouped in multi-variable files, which name cannot be computed directly from datasets attributes alone but requires to use an extra_facets file, which principles are explained in *Extra Facets*, and which content is `available here`. These multi-variable files must also undergo some data selection.

13.6 Use of extra facets in fixes

Extra facets are a mechanism to provide additional information for certain kinds of data. The general approach is described in *Extra Facets*. Here, we describe how they can be used in fixes to mold data into the form required by the applicable standard. For example, if the input data is part of an observational product that delivers surface temperature with a variable name of *t2m* inside a file named *2m_temperature_1950_monthly.nc*, but the same variable is called *tas* in the applicable standard, a fix can be created that reads the original variable from the correct file, and provides a renamed variable to the rest of the processing chain.

Normally, the applicable standard for variables is CMIP6.

For more details, refer to existing uses of this feature as examples, as e.g. *for IPSL-CM6*.

CHAPTER

FOURTEEN

DERIVING A VARIABLE

The variable derivation preprocessor module allows to derive variables which are not in the CMIP standard data request using standard variables as input. This is a special type of *preprocessor function*. All derivation scripts are located in esmvalcore/preprocessor/_derive/. A typical example looks like this:

```
"""Derivation of variable `dummy`."""
from ._baseclass import DerivedVariableBase


class DerivedVariable(DerivedVariableBase):
    """Derivation of variable `dummy`."""

    @staticmethod
    def required(project):
        """Declare the variables needed for derivation."""
        mip = 'fx'
        if project == 'CMIP6':
            mip = 'Ofx'
        required = [
            {'short_name': 'var_a'},
            {'short_name': 'var_b', 'mip': mip, 'optional': True},
        ]
        return required

    @staticmethod
    def calculate(cubes):
        """Compute `dummy`."""

        # `cubes` is a CubeList containing all required variables.
        cube = do_something_with(cubes)

        # Return single cube at the end
        return cube
```

The static function `required(project)` returns a `list` of `dict` containing all required variables for deriving the derived variable. Its only argument is the `project` of the specific dataset. In this particular example script, the derived variable `dummy` is derived from `var_a` and `var_b`. It is possible to specify arbitrary attributes for each required variable, e.g. `var_b` uses the mip `fx` (or `Ofx` in the case of CMIP6) instead of the original one of `dummy`. Note that you can also declare a required variable as `optional=True`, which allows the skipping of this particular variable during data extraction. For example, this is useful for fx variables which are often not available for observational datasets. Otherwise, the tool will fail if not all required variables are available for all datasets.

The actual derivation takes place in the static function `calculate(cubes)` which returns a single `cube` containing

the derived variable. Its only argument `cubes` is a `CubeList` containing all required variables.

Part V

Contributions are very welcome

We greatly value contributions of any kind. Contributions could include, but are not limited to documentation improvements, bug reports, new or improved code, scientific and technical code reviews, infrastructure improvements, mailing list and chat participation, community help/building, education and outreach. We value the time you invest in contributing and strive to make the process as easy as possible. If you have suggestions for improving the process of contributing, please do not hesitate to propose them.

If you have a bug or other issue to report or just need help, please open an issue on the issues tab on the ESMValCore github repository.

If you would like to contribute a new *preprocessor function*, *derived variable*, *fix for a dataset*, or another new feature, please discuss your idea with the development team before getting started, to avoid double work and/or disappointment later. A good way to do this is to open an issue on GitHub.

CHAPTER

FIFTEEN

GETTING STARTED

See *Installation from source* for instructions on how to set up a development installation.

New development should preferably be done in the ESMValCore GitHub repository. The default git branch is `main`. Use this branch to create a new feature branch from and make a pull request against. This page offers a good introduction to git branches, but it was written for BitBucket while we use GitHub, so replace the word BitBucket by GitHub whenever you read it.

It is recommended that you open a draft pull request early, as this will cause *CircleCI to run the unit tests*, *Codacy to analyse your code*, and *readthedocs to build the documentation*. It's also easier to get help from other developers if your code is visible in a pull request.

Make small pull requests, the ideal pull requests changes just a few files and adds/changes no more than 100 lines of production code. The amount of test code added can be more extensive, but changes to existing test code should be made sparingly.

15.1 Design considerations

When making changes, try to respect the current structure of the program. If you need to make major changes to the structure of program to add a feature, chances are that you have either not come up with the most optimal design or the feature is not a very good fit for the tool. Discuss your feature with the @ESMValGroup/esmvaltool-coreteam in an issue to find a solution.

Please keep the following considerations in mind when programming:

- Changes should preferably be *backward compatible*.
- Apply changes gradually and change no more than a few files in a single pull request, but do make sure every pull request in itself brings a meaningful improvement. This reduces the risk of breaking existing functionality and making *backward incompatible* changes, because it helps you as well as the reviewers of your pull request to better understand what exactly is being changed.
- *Preprocessor functions* are Python functions (and not classes) so they are easy to understand and implement for scientific contributors.
- No additional CMOR checks should be implemented inside preprocessor functions. The input cube is fixed and confirmed to follow the specification in esmvalcore/cmor/tables before applying any other preprocessor functions. This design helps to keep the preprocessor functions and diagnostics scripts that use the preprocessed data from the tool simple and reliable. See *Project CMOR table configuration* for the mapping from `project` in the recipe to the relevant CMOR table.
- The ESMValCore package is based on iris. Preprocessor functions should preferably be small and just call the relevant iris code. Code that is more involved and more broadly applicable than just in the ESMValCore, should be implemented in iris instead.

- Any settings in the recipe that can be checked before loading the data should be checked at the *task creation stage*. This avoids that users run a recipe for several hours before finding out they made a mistake in the recipe. No data should be processed or files written while creating the tasks.
- CMOR checks should provide a good balance between reliability of the tool and ease of use. Several *levels of strictness of the checks* are available to facilitate this.
- Keep your code short and simple: we would like to make contributing as easy as possible. For example, avoid implementing complicated class inheritance structures and boilerplate code.
- If you find yourself copy-pasting a piece of code and making minor changes to every copy, instead put the repeated bit of code in a function that you can re-use, and provide the changed bits as function arguments.
- Be careful when changing existing unit tests to make your new feature work. You might be breaking existing features if you have to change existing tests.

Finally, if you would like to improve the design of the tool, discuss your plans with the @ESMValGroup/esmvaltool-coreteam to make sure you understand the current functionality and you all agree on the new design.

CHAPTER

SIXTEEN

CHECKLIST FOR PULL REQUESTS

To clearly communicate up front what is expected from a pull request, we have the following checklist. Please try to do everything on the list before requesting a review. If you are unsure about something on the list, please ask the @ESMValGroup/tech-reviewers or @ESMValGroup/science-reviewers for help by commenting on your (draft) pull request or by starting a new discussion.

In the ESMValTool community we use pull request reviews to ensure all code and documentation contributions are of good quality. The icons indicate whether the item will be checked during the Technical review or Scientific review.

- The new functionality is *relevant and scientifically sound*
- *The pull request has a descriptive title and labels*
- Code is written according to the *code quality guidelines*
- and *Documentation* is available
- Unit *tests* have been added
- Changes are *backward compatible*
- Changed *dependencies have been added or removed correctly*
- The *list of authors* is up to date
- The *checks shown below the pull request* are successful

CHAPTER

SEVENTEEN

SCIENTIFIC RELEVANCE

The proposed changes should be relevant for the larger scientific community. The implementation of new features should be scientifically sound; e.g. the formulas used in new preprocesssor functions should be accompanied by the relevant references and checked for correctness by the scientific reviewer. The CF Conventions as well as additional standards imposed by CMIP should be followed whenever possible.

CHAPTER

EIGHTEEN

PULL REQUEST TITLE AND LABEL

The title of a pull request should clearly describe what the pull request changes. If you need more text to describe what the pull request does, please add it in the description. Add one or more labels to your pull request to indicate the type of change. At least one of the following labels should be used: *bug*, *deprecated feature*, *fix for dataset*, *preprocessor*, *cmor*, *api*, *testing*, *documentation* or *enhancement*.

The titles and labels of pull requests are used to compile the Changelog, therefore it is important that they are easy to understand for people who are not familiar with the code or people in the project. Descriptive pull request titles also makes it easier to find back what was changed when, which is useful in case a bug was introduced.

CHAPTER

NINETEEN

CODE QUALITY

To increase the readability and maintainability or the ESMValCore source code, we aim to adhere to best practices and coding standards.

We include checks for Python and yaml files, most of which are described in more detail in the sections below. This includes checks for invalid syntax and formatting errors. Pre-commit is a handy tool that can run all of these checks automatically just before you commit your code. It knows knows which tool to run for each filetype, and therefore provides a convenient way to check your code.

19.1 Python

The standard document on best practices for Python code is PEP8 and there is PEP257 for code documentation. We make use of numpy style docstrings to document Python functions that are visible on readthedocs.

To check if your code adheres to the standard, go to the directory where the repository is cloned, e.g. `cd ESMValCore`, and run prospector

```
prospector esmvalcore/preprocessor/_regrid.py
```

In addition to prospector, we use flake8 to automatically check for bugs and formatting mistakes and mypy for checking that type hints are correct. Note that type hints are completely optional, but if you do choose to add them, they should be correct.

When you make a pull request, adherence to the Python development best practices is checked in two ways:

1. As part of the unit tests, flake8 and mypy are run by CircleCI, see the section on *Tests* for more information.
2. Codacy is a service that runs prospector (and other code quality tools) on changed files and reports the results. Click the 'Details' link behind the Codacy check entry and then click 'View more details on Codacy Production' to see the results of the static code analysis done by Codacy. If you need to log in, you can do so using your GitHub account.

The automatic code quality checks by prospector are really helpful to improve the quality of your code, but they are not flawless. If you suspect prospector or Codacy may be wrong, please ask the @ESMValGroup/tech-reviewers by commenting on your pull request.

Note that running prospector locally will give you quicker and sometimes more accurate results than waiting for Codacy.

Most formatting issues in Python code can be fixed automatically by running the commands

```
isort some_file.py
```

to sort the imports in the standard way using isort and

```
yapf -i some_file.py
```

to add/remove whitespace as required by the standard using yapf,

```
docformatter -i some_file.py
```

to run docformatter which helps formatting the docstrings (such as line length, spaces).

19.2 YAML

Please use yamllint to check that your YAML files do not contain mistakes. `yamllint` checks for valid syntax, common mistakes like key repetition and cosmetic problems such as line length, trailing spaces, wrong indentation, etc.

19.3 Any text file

A generic tool to check for common spelling mistakes is codespell.

CHAPTER

TWENTY

DOCUMENTATION

The documentation lives on docs.esmvaltool.org.

20.1 Adding documentation

The documentation is built by readthedocs using Sphinx. There are two main ways of adding documentation:

1. As written text in the directory doc. When writing reStructuredText (`.rst`) files, please try to limit the line length to 80 characters and always start a sentence on a new line. This makes it easier to review changes to documentation on GitHub.
2. As docstrings or comments in code. For Python code, only the docstrings of Python modules, classes, and functions that are mentioned in doc/api are used to generate the online documentation. This results in the *ESMValCore API Reference*. The standard document with best practices on writing docstrings is PEP257. For the API documentation, we make use of numpy style docstrings.

20.2 What should be documented

Functionality that is visible to users should be documented. Any documentation that is visible on readthedocs should be well written and adhere to the standards for documentation. Examples of this include:

- The *recipe*
- Preprocessor *functions* and their *use from the recipe*
- *Configuration options*
- *Installation*
- *Output files*
- *Command line interface*
- *Diagnostic script interfaces*
- *The experimental Python interface*

Note that:

- For functions that compute scientific results, comments with references to papers and/or other resources as well as formula numbers should be included.
- When making changes to/introducing a new preprocessor function, also update the *preprocessor documentation*.

- There is no need to write complete numpy style documentation for functions that are not visible in the *ESMValCore API Reference* chapter on readthedocs. However, adding a docstring describing what a function does is always a good idea. For short functions, a one-line docstring is usually sufficient, but more complex functions might require slightly more extensive documentation.

When reviewing a pull request, always check that documentation is easy to understand and available in all expected places.

20.3 How to build and view the documentation

Whenever you make a pull request or push new commits to an existing pull request, readthedocs will automatically build the documentation. The link to the documentation will be shown in the list of checks below your pull request. Click 'Details' behind the check `docs/readthedocs.org:esmvaltool` to preview the documentation. If all checks were successful, you may need to click 'Show all checks' to see the individual checks.

To build the documentation on your own computer, go to the directory where the repository was cloned and run

```
python setup.py build_sphinx
```

or

```
python setup.py build_sphinx -Ea
```

to build it from scratch.

Make sure that your newly added documentation builds without warnings or errors and looks correctly formatted. CircleCI will build the documentation with the command:

```
python setup.py build_sphinx --warning-is-error
```

This will catch mistakes that can be detected automatically.

The configuration file for Sphinx is doc/shinx/source/conf.py.

See Integration with the ESMValCore documentation for information on how the ESMValCore documentation is integrated into the complete ESMValTool project documentation on readthedocs.

When reviewing a pull request, always check that the documentation checks shown below the pull request were successful.

CHAPTER

TWENTYONE

TESTS

To check that the code works correctly, there tests available in the tests directory. We use pytest to write and run our tests.

Contributions to ESMValCore should be covered by unit tests. Have a look at the existing tests in the `tests` directory for inspiration on how to write your own tests. If you do not know how to start with writing unit tests, ask the @ESMValGroup/tech-reviewers for help by commenting on the pull request and they will try to help you. To check which parts of your code are covered by tests, open the file `test-reports/coverage_html/index.html` and browse to the relevant file. It is also possible to view code coverage on Codacy (click the Files tab) and CircleCI (open the `tests` job and click the ARTIFACTS tab).

Whenever you make a pull request or push new commits to an existing pull request, the tests in the tests directory of the branch associated with the pull request will be run automatically on CircleCI. The results appear at the bottom of the pull request. Click on 'Details' for more information on a specific test job. To see some of the results on CircleCI, you may need to log in. You can do so using your GitHub account.

To run the tests on your own computer, go to the directory where the repository is cloned and run the command

```
pytest
```

Optionally you can skip tests which require additional dependencies for supported diagnostic script languages by adding `-m 'not installation'` to the previous command.

When reviewing a pull request, always check that all test jobs on CircleCI were successful.

21.1 Sample data

New or modified preprocessor functions should preferably also be tested using the sample data. These tests are located in tests/sample_data. Please mark new tests that use the sample data with the decorator `@pytest.mark.use_sample_data`.

The ESMValTool_sample_data repository contains samples of CMIP6 data for testing ESMValCore. The ESMValTool-sample-data package is installed as part of the developer dependencies. The size of the package is relatively small (~ 100 MB), so it can be easily downloaded and distributed.

Preprocessing the sample data can be time-consuming, so some intermediate results are cached by pytest to make the tests run faster. If you suspect the tests are failing because the cache is invalid, clear it by running

```
pytest --cache-clear
```

To avoid running the time consuming tests that use sample data altogether, run

```
pytest -m "not use_sample_data"
```

21.2 Automated testing

Whenever you make a pull request or push new commits to an existing pull request, the tests in the tests directory of the branch associated with the pull request will be run automatically on CircleCI.

Every night, more extensive tests are run to make sure that problems with the installation of the tool are discovered by the development team before users encounter them. These nightly tests have been designed to follow the installation procedures described in the documentation, e.g. in the *Installation* chapter. The nightly tests are run using both CircleCI and GitHub Actions. The result of the tests ran by CircleCI can be seen on the CircleCI project page and the result of the tests ran by GitHub Actions can be viewed on the Actions tab of the repository.

The configuration of the tests run by CircleCI can be found in the directory .circleci, while the configuration of the tests run by GitHub Actions can be found in the directory .github/workflows.

CHAPTER

TWENTYTWO

BACKWARD COMPATIBILITY

The ESMValCore package is used by many people to run their recipes. Many of these recipes are maintained in the public ESMValTool repository, but there are also users who choose not to share their work there. While our commitment is first and foremost to users who do share their recipes in the ESMValTool repository, we still try to be nice to all of the ESMValCore users. When making changes, e.g. to the *recipe format*, the *diagnostic script interface*, the public *Python API*, or the *configuration file format*, keep in mind that this may affect many users. To keep the tool user friendly, try to avoid making changes that are not backward compatible, i.e. changes that require users to change their existing recipes, diagnostics, configuration files, or scripts.

If you really must change the public interfaces of the tool, always discuss this with the @ESMValGroup/esmvaltool-coreteam. Try to deprecate the feature first by issuing a `DeprecationWarning` using the `warnings` module and schedule it for removal three minor versions from the latest released version. For example, when you deprecate a feature in a pull request that will be included in version 2.3, that feature could be removed in version 2.5. Mention the version in which the feature will be removed in the deprecation message. Label the pull request with the deprecated feature label. When deprecating a feature, please follow up by actually removing the feature in due course.

If you must make backward incompatible changes, you need to update the available recipes in ESMValTool and link the ESMValTool pull request(s) in the ESMValCore pull request description. You can ask the @ESMValGroup/esmvaltool-recipe-maintainers for help with updating existing recipes, but please be considerate of their time.

When reviewing a pull request, always check for backward incompatible changes and make sure they are needed and have been discussed with the @ESMValGroup/esmvaltool-coreteam. Also, make sure the author of the pull request has created the accompanying pull request(s) to update the ESMValTool, before merging the ESMValCore pull request.

CHAPTER

TWENTYTHREE

DEPENDENCIES

Before considering adding a new dependency, carefully check that the license of the dependency you want to add and any of its dependencies are compatible with the Apache 2.0 license that applies to the ESMValCore. Note that GPL version 2 license is considered incompatible with the Apache 2.0 license, while the compatibility of GPL version 3 license with the Apache 2.0 license is questionable. See this statement by the authors of the Apache 2.0 license for more information.

When adding or removing dependencies, please consider applying the changes in the following files:

- `environment.yml` contains development dependencies that cannot be installed from PyPI
- `docs/requirements.txt` contains Python dependencies needed to build the documentation that can be installed from PyPI
- `docs/conf.py` contains a list of Python dependencies needed to build the documentation that cannot be installed from PyPI and need to be mocked when building the documentation. (We do not use conda to build the documentation because this is too time consuming.)
- `setup.py` contains all Python dependencies, regardless of their installation source
- `package/meta.yaml` contains dependencies for the conda package; all Python and compiled dependencies that can be installed from conda should be listed here

Note that packages may have a different name on conda-forge than on PyPI.

Several test jobs on CircleCI related to the installation of the tool will only run if you change the dependencies. These will be skipped for most pull requests.

When reviewing a pull request where dependencies are added or removed, always check that the changes have been applied in all relevant files.

CHAPTER

TWENTYFOUR

LIST OF AUTHORS

If you make a contribution to ESMValCore and you would like to be listed as an author (e.g. on Zenodo), please add your name to the list of authors in `CITATION.cff` and generate the entry for the `.zenodo.json` file by running the commands

```
pip install cffconvert
cffconvert --ignore-suspect-keys --outputformat zenodo --outfile .zenodo.json
```

Presently, this method unfortunately discards entries *communities* and *grants* from that file; please restore them manually, or alternately proceed with the addition manually

CHAPTER

TWENTYFIVE

PULL REQUEST CHECKS

To check that a pull request is up to standard, several automatic checks are run when you make a pull request. Read more about it in the *Tests* and *Documentation* sections. Successful checks have a green ✓ in front, a means the check failed.

If you need help with the checks, please ask the technical reviewer of your pull request for help. Ask @ESMValGroup/tech-reviewers if you do not have a technical reviewer yet.

If the checks are broken because of something unrelated to the current pull request, please check if there is an open issue that reports the problem. Create one if there is no issue yet. You can attract the attention of the @ESMValGroup/esmvaltool-coreteam by mentioning them in the issue if it looks like no-one is working on solving the problem yet. The issue needs to be fixed in a separate pull request first. After that has been merged into the `main` branch and all checks on this branch are green again, merge it into your own branch to get the tests to pass.

When reviewing a pull request, always make sure that all checks were successful. If the Codacy check keeps failing, please run prospector locally. If necessary, ask the pull request author to do the same and to address the reported issues. See the section on *code_quality* for more information. Never merge a pull request with failing CircleCI or readthedocs checks.

CHAPTER

TWENTYSIX

MAKING A RELEASE

The release manager makes the release, assisted by the release manager of the previous release, or if that person is not available, another previous release manager. Perform the steps listed below with two persons, to reduce the risk of error.

To make a new release of the package, follow these steps:

26.1 1. Check the tests on GitHub Actions and CircleCI

Check the `nightly` build on CircleCI and the GitHub Actions run. All tests should pass before making a release (branch).

26.2 2. Create a release branch

Create a branch off the `main` branch and push it to GitHub. Ask someone with administrative permissions to set up branch protection rules for it so only you and the person helping you with the release can push to it. Announce the name of the branch in an issue and ask the members of the ESMValTool development team to run their favourite recipe using this branch.

26.3 3. Increase the version number

The version number is stored in `esmvalcore/_version.py`, `package/meta.yaml`, `CITATION.cff`. Make sure to update all files. Also update the release date in `CITATION.cff`. See https://semver.org for more information on choosing a version number. Make a pull request and get it merged into `main` and cherry pick it into the release branch.

26.4 4. Add release notes

Use the script esmvaltool/utils/draft_release_notes.py to create create a draft of the release notes. This script uses the titles and labels of merged pull requests since the previous release. Review the results, and if anything needs changing, change it on GitHub and re-run the script until the changelog looks acceptable. Copy the result to the file `doc/changelog.rst`. Make a pull request and get it merged into `main` and cherry pick it into the release branch..

26.5 5. Cherry pick bugfixes into the release branch

If a bug is found and fixed (i.e. pull request merged into the `main` branch) during the period of testing, use the command `git cherry-pick` to include the commit for this bugfix into the release branch. When the testing period is over, make a pull request to update the release notes with the latest changes, get it merged into `main` and cherry-pick it into the release branch.

26.6 6. Make the release on GitHub

Do a final check that all tests on CircleCI and GitHub Actions completed successfully. Then click the releases tab and create the new release from the release branch (i.e. not from `main`).

26.7 7. Create and upload the Conda package

The package is automatically uploaded to the ESMValGroup conda channel by a GitHub action. If this has failed for some reason, build and upload the package manually by following the instructions below.

Follow these steps to create a new conda package:

- Check out the tag corresponding to the release, e.g. `git checkout tags/v2.1.0`
- Make sure your current working directory is clean by checking the output of `git status` and by running `git clean -xdf` to remove any files ignored by git.
- Edit `package/meta.yaml` and uncomment the lines starting with `git_rev` and `git_url`, remove the line starting with `path` in the `source` section.
- Activate the base environment `conda activate base`
- Install the required packages: `conda install -y conda-build conda-verify ripgrep anaconda-client`
- Run `conda build package -c conda-forge` to build the conda package
- If the build was successful, upload the package to the esmvalgroup conda channel, e.g. `anaconda upload --user esmvalgroup /path/to/conda/conda-bld/noarch/esmvalcore-2.3.0-py_0.tar.bz2`.

26.8 8. Create and upload the PyPI package

The package is automatically uploaded to the PyPI by a GitHub action. If has failed for some reason, build and upload the package manually by following the instructions below.

Follow these steps to create a new Python package:

- Check out the tag corresponding to the release, e.g. `git checkout tags/v2.1.0`
- Make sure your current working directory is clean by checking the output of `git status` and by running `git clean -xdf` to remove any files ignored by git.
- Install the required packages: `python3 -m pip install --upgrade pep517 twine`
- Build the package: `python3 -m pep517.build --source --binary --out-dir dist/ .` This command should generate two files in the `dist` directory, e.g. `ESMValCore-2.3.0-py3-none-any.whl` and `ESMValCore-2.3.0.tar.gz`.

- Upload the package: `python3 -m twine upload dist/*` You will be prompted for an API token if you have not set this up before, see here for more information.

You can read more about this in Packaging Python Projects.

Part VI

ESMValCore API Reference

ESMValCore is mostly used as a commandline tool. However, it is also possibly to use (parts of) ESMValTool as a library. This section documents the public API of ESMValCore.

CHAPTER

TWENTYSEVEN

CMOR FUNCTIONS

CMOR module.

27.1 Checking compliance

Module for checking iris cubes against their CMOR definitions.

Classes:

CMORCheck(cube, var_info[, frequency, ...])	Class used to check the CMOR-compliance of the data.
CheckLevels(value)	Level of strictness of the checks.

Exceptions:

CMORCheckError	Exception raised when a cube does not pass the CMORCheck.

Functions:

cmor_check(cube, cmor_table, mip, ...)	Check if cube conforms to variable's CMOR definition.
cmor_check_data(cube, cmor_table, mip, ...)	Check if data conforms to variable's CMOR definition.
cmor_check_metadata(cube, cmor_table, mip, ...)	Check if metadata conforms to variable's CMOR definition.

class esmvalcore.cmor.check.**CMORCheck**(*cube*, *var_info*, *frequency=None*, *fail_on_error=False*, *check_level=<CheckLevels.DEFAULT: 3>*, *automatic_fixes=False*)

Bases: object

Class used to check the CMOR-compliance of the data.

It can also fix some minor errors and does some minor data homogeneization:

Parameters

- **cube** (*iris.cube.Cube:*) – Iris cube to check.
- **var_info** (*variables_info.VariableInfo*) – Variable info to check.
- **frequency** (*str*) – Expected frequency for the data.
- **fail_on_error** (*bool*) – If true, CMORCheck stops on the first error. If false, it collects

all possible errors before stopping.

- **automatic_fixes** (*bool*) – If True, CMORCheck will try to apply automatic fixes for any detected error, if possible.
- **check_level** (CheckLevels) – Level of strictness of the checks.

frequency
Expected frequency for the data.

Type str

Methods:

check_data([logger])	Check the cube data.
check_metadata([logger])	Check the cube metadata.
has_debug_messages()	Check if there are reported debug messages.
has_errors()	Check if there are reported errors.
has_warnings()	Check if there are reported warnings.
report(level, message, *args)	Report a message from the checker.
report_critical(message, *args)	Report an error.
report_debug_message(message, *args)	Report a debug message.
report_debug_messages()	Report detected debug messages to the given logger.
report_error(message, *args)	Report a normal error.
report_errors()	Report detected errors.
report_warning(message, *args)	Report a warning level error.
report_warnings()	Report detected warnings to the given logger.

check_data(*logger=None*)
Check the cube data.

Performs all the tests that require to have the data in memory. Assumes that metadata is correct, so you must call check_metadata prior to this.

It will also report some warnings in case of minor errors.

Parameters **logger** (*logging.Logger*) – Given logger.

Raises ***CMORCheckError*** – If errors are found. If fail_on_error attribute is set to True, raises as soon as an error is detected. If set to False, it perform all checks and then raises.

check_metadata(*logger=None*)
Check the cube metadata.

Perform all the tests that do not require to have the data in memory.

It will also report some warnings in case of minor errors and homogenize some data:

- Equivalent calendars will all default to the same name.
- Time units will be set to days since 1850-01-01

Parameters **logger** (*logging.Logger*) – Given logger.

Raises ***CMORCheckError*** – If errors are found. If fail_on_error attribute is set to True, raises as soon as an error is detected. If set to False, it perform all checks and then raises.

has_debug_messages()
Check if there are reported debug messages.

Returns True if there are pending debug messages, False otherwise.

Return type bool

has_errors()
Check if there are reported errors.

Returns True if there are pending errors, False otherwise.

Return type bool

has_warnings()
Check if there are reported warnings.

Returns True if there are pending warnings, False otherwise.

Return type bool

report(*level*, *message*, **args*)
Report a message from the checker.

Parameters

- **level** (CheckLevels) – Message level
- **message** (*str*) – Message to report
- **args** – String format args for the message

Raises *CMORCheckError* – If fail on error is set, it is thrown when registering an error message

report_critical(*message*, **args*)
Report an error.

If fail_on_error is set to True, raises automatically. If fail_on_error is set to False, stores it for later reports.

Parameters

- **message** (*str: unicode*) – Message for the error.
- ***args** – arguments to format the message string.

report_debug_message(*message*, **args*)
Report a debug message.

Parameters

- **message** (*str: unicode*) – Message for the debug logger.
- ***args** – arguments to format the message string

report_debug_messages()
Report detected debug messages to the given logger.

Parameters **logger** (*logging.Logger*) – Given logger.

report_error(*message*, **args*)
Report a normal error.

Parameters

- **message** (*str: unicode*) – Message for the error.
- ***args** – arguments to format the message string.

report_errors()
Report detected errors.

Raises *CMORCheckError* – If any errors were reported before calling this method.

report_warning(*message*, **args*)
Report a warning level error.

Parameters

- **message** (*str: unicode*) – Message for the warning.
- ***args** – arguments to format the message string.

report_warnings()
Report detected warnings to the given logger.

Parameters **logger** (*logging.Logger*) – Given logger

exception esmvalcore.cmor.check.**CMORCheckError**
Bases: Exception

Exception raised when a cube does not pass the CMORCheck.

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

class esmvalcore.cmor.check.**CheckLevels**(*value*)
Bases: enum.IntEnum

Level of strictness of the checks.

- DEBUG

Type Report any debug message that the checker wants to communicate.

- STRICT

Type Fail if there are warnings regarding compliance of CMOR standards.

- DEFAULT

Type Fail if cubes present any discrepancy with CMOR standards.

- RELAXED

Type Fail if cubes present severe discrepancies with CMOR standards.

- IGNORE

Type Do not fail for any discrepancy with CMOR standards.

Attributes:

DEBUG
DEFAULT
IGNORE
RELAXED
STRICT

DEBUG = 1

DEFAULT = 3

IGNORE = 5

RELAXED = 4

STRICT = 2

esmvalcore.cmor.check.**cmor_check**(*cube*, *cmor_table*, *mip*, *short_name*, *frequency*, *check_level*)

Check if cube conforms to variable's CMOR definition.

Equivalent to calling cmor_check_metadata and cmor_check_data consecutively.

Parameters

- **cube** (*iris.cube.Cube*) – Data cube to check.
- **cmor_table** (*str*) – CMOR definitions to use.
- **mip** – Variable's mip.
- **short_name** (*str*) – Variable's short name.
- **frequency** (*str*) – Data frequency.
- **check_level** (*enum.IntEnum*) – Level of strictness of the checks.

esmvalcore.cmor.check.**cmor_check_data**(*cube*, *cmor_table*, *mip*, *short_name*, *frequency*, *check_level=<CheckLevels.DEFAULT: 3>*)

Check if data conforms to variable's CMOR definition.

The checks performed at this step require the data in memory.

Parameters

- **cube** (*iris.cube.Cube*) – Data cube to check.
- **cmor_table** (*str*) – CMOR definitions to use.
- **mip** – Variable's mip.
- **short_name** (*str*) – Variable's short name
- **frequency** (*str*) – Data frequency
- **check_level** (CheckLevels) – Level of strictness of the checks.

esmvalcore.cmor.check.**cmor_check_metadata**(*cube*, *cmor_table*, *mip*, *short_name*, *frequency*, *check_level=<CheckLevels.DEFAULT: 3>*)

Check if metadata conforms to variable's CMOR definition.

None of the checks at this step will force the cube to load the data.

Parameters

- **cube** (*iris.cube.Cube*) – Data cube to check.
- **cmor_table** (*str*) – CMOR definitions to use.
- **mip** – Variable's mip.
- **short_name** (*str*) – Variable's short name.
- **frequency** (*str*) – Data frequency.
- **check_level** (CheckLevels) – Level of strictness of the checks.

27.2 Automatically fixing issues

Apply automatic fixes for known errors in cmorized data.

All functions in this module will work even if no fixes are available for the given dataset. Therefore is recommended to apply them to all variables to be sure that all known errors are fixed.

Functions:

fix_data(cube, short_name, project, dataset, mip)	Fix cube data if fixes add present and check it anyway.
fix_file(file, short_name, project, dataset, ...)	Fix files before ESMValTool can load them.
fix_metadata(cubes, short_name, project, ...)	Fix cube metadata if fixes are required and check it anyway.

esmvalcore.cmor.fix.**fix_data**(*cube*, *short_name*, *project*, *dataset*, *mip*, *frequency=None*, *check_level=<CheckLevels.DEFAULT: 3>*, ***extra_facets*)

Fix cube data if fixes add present and check it anyway.

This method assumes that metadata is already fixed and checked.

This method collects all the relevant fixes for a given variable, applies them and checks resulting cube (or the original if no fixes were needed) metadata to ensure that it complies with the standards of its project CMOR tables.

Parameters

- **cube** (*iris.cube.Cube*) – Cube to fix
- **short_name** (*str*) – Variable's short name
- **project** (*str*) –
- **dataset** (*str*) –
- **mip** (*str*) – Variable's MIP
- **frequency** (*str, optional*) – Variable's data frequency, if available
- **check_level** (CheckLevels) – Level of strictness of the checks. Set to default.
- ****extra_facets** (*dict, optional*) – Extra facets are mainly used for data outside of the big projects like CMIP, CORDEX, obs4MIPs. For details, see *Extra Facets*.

Returns Fixed and checked cube

Return type iris.cube.Cube

Raises ***CMORCheckError*** – If the checker detects errors in the data that it can not fix.

esmvalcore.cmor.fix.**fix_file**(*file*, *short_name*, *project*, *dataset*, *mip*, *output_dir*, ***extra_facets*)

Fix files before ESMValTool can load them.

This fixes are only for issues that prevent iris from loading the cube or that cannot be fixed after the cube is loaded.

Original files are not overwritten.

Parameters

- **file** (*str*) – Path to the original file
- **short_name** (*str*) – Variable's short name
- **project** (*str*) –

- **dataset** (*str*) –
- **output_dir** (*str*) – Output directory for fixed files
- ****extra_facets** (*dict, optional*) – Extra facets are mainly used for data outside of the big projects like CMIP, CORDEX, obs4MIPs. For details, see *Extra Facets*.

Returns Path to the fixed file

Return type str

esmvalcore.cmor.fix.**fix_metadata**(*cubes*, *short_name*, *project*, *dataset*, *mip*, *frequency=None*, *check_level=<CheckLevels.DEFAULT: 3>*, ***extra_facets*)

Fix cube metadata if fixes are required and check it anyway.

This method collects all the relevant fixes for a given variable, applies them and checks the resulting cube (or the original if no fixes were needed) metadata to ensure that it complies with the standards of its project CMOR tables.

Parameters

- **cubes** (*iris.cube.CubeList*) – Cubes to fix
- **short_name** (*str*) – Variable's short name
- **project** (*str*) –
- **dataset** (*str*) –
- **mip** (*str*) – Variable's MIP
- **frequency** (*str, optional*) – Variable's data frequency, if available
- **check_level** (CheckLevels) – Level of strictness of the checks. Set to default.
- ****extra_facets** (*dict, optional*) – Extra facets are mainly used for data outside of the big projects like CMIP, CORDEX, obs4MIPs. For details, see *Extra Facets*.

Returns Fixed and checked cube

Return type iris.cube.Cube

Raises ***CMORCheckError*** – If the checker detects errors in the metadata that it can not fix.

27.3 Functions for fixing issues

Functions for fixing specific issues with datasets.

Functions:

add_altitude_from_plev(cube)	Add altitude coordinate from pressure level coordinate.
add_plev_from_altitude(cube)	Add pressure level coordinate from altitude coordinate.
add_sigma_factory(cube)	Add factory for `atmosphere_sigma_coordinate`.

esmvalcore.cmor.fixes.**add_altitude_from_plev**(*cube*)

Add altitude coordinate from pressure level coordinate.

Parameters **cube** (*iris.cube.Cube*) – Input cube.

Raises **ValueError** – `cube` does not contain coordinate `air_pressure`.

esmvalcore.cmor.fixes.**add_plev_from_altitude**(*cube*)

Add pressure level coordinate from altitude coordinate.

Parameters **cube** (*iris.cube.Cube*) – Input cube.

Raises **ValueError** – cube does not contain coordinate `altitude`.

`esmvalcore.cmor.fixes.`**add_sigma_factory**(*cube*)
Add factory for `atmosphere_sigma_coordinate`.

Parameters **cube** (*iris.cube.Cube*) – Input cube.

Raises **ValueError** – cube does not contain coordinate `atmosphere_sigma_coordinate`.

27.4 Using CMOR tables

CMOR information reader for ESMValTool.

Read variable information from CMOR 2 and CMOR 3 tables and make it easily available for the other components of ESMValTool

Classes:

CMIP3Info(cmor_tables_path[, default, ...])	Class to read CMIP3-like data request.
CMIP5Info(cmor_tables_path[, default, ...])	Class to read CMIP5-like data request.
CMIP6Info(cmor_tables_path[, default, ...])	Class to read CMIP6-like data request.
CoordinateInfo(name)	Class to read and store coordinate information.
CustomInfo([cmor_tables_path])	Class to read custom var info for ESMVal.
InfoBase(default, alt_names, strict)	Base class for all table info classes.
JsonInfo()	Base class for the info classes.
TableInfo(*args, **kwargs)	Container class for storing a CMOR table.
VariableInfo(table_type, short_name)	Class to read and store variable information.

Data:

CMOR_TABLES	CMOR info objects.

Functions:

get_var_info(project, mip, short_name)	Get variable information.
read_cmor_tables([cfg_developer])	Read cmor tables required in the configuration.

class `esmvalcore.cmor.table.`**CMIP3Info**(*cmor_tables_path*, *default=None*, *alt_names=None*, *strict=True*)
Bases: *esmvalcore.cmor.table.CMIP5Info*

Class to read CMIP3-like data request.

Parameters

- **cmor_tables_path** (*str*) – Path to the folder containing the Tables folder with the json files
- **default** (*object*) – Default table to look variables on if not found
- **strict** (*bool*) – If False, will look for a variable in other tables if it can not be found in the requested one

Methods:

get_table(table)	Search and return the table info.
get_variable(table_name, short_name[, derived])	Search and return the variable info.

get_table(*table*)
Search and return the table info.

Parameters **table** (*str*) – Table name

Returns Return the TableInfo object for the requested table if found, returns None if not

Return type *TableInfo*

get_variable(*table_name*, *short_name*, *derived=False*)
Search and return the variable info.

Parameters

- **table_name** (*str*) – Table name
- **short_name** (*str*) – Variable's short name
- **derived** (*bool, optional*) – Variable is derived. Info retrieval for derived variables always look on the default tables if variable is not find in the requested table

Returns Return the VariableInfo object for the requested variable if found, returns None if not

Return type *VariableInfo*

class esmvalcore.cmor.table.**CMIP5Info**(*cmor_tables_path*, *default=None*, *alt_names=None*, *strict=True*)
Bases: *esmvalcore.cmor.table.InfoBase*

Class to read CMIP5-like data request.

Parameters

- **cmor_tables_path** (*str*) – Path to the folder containing the Tables folder with the json files
- **default** (*object*) – Default table to look variables on if not found
- **strict** (*bool*) – If False, will look for a variable in other tables if it can not be found in the requested one

Methods:

get_table(table)	Search and return the table info.
get_variable(table_name, short_name[, derived])	Search and return the variable info.

get_table(*table*)
Search and return the table info.

Parameters **table** (*str*) – Table name

Returns Return the TableInfo object for the requested table if found, returns None if not

Return type *TableInfo*

get_variable(*table_name*, *short_name*, *derived=False*)
Search and return the variable info.

Parameters

- **table_name** (*str*) – Table name

- **short_name** (*str*) – Variable's short name
- **derived** (*bool, optional*) – Variable is derived. Info retrieval for derived variables always look on the default tables if variable is not find in the requested table

Returns Return the VariableInfo object for the requested variable if found, returns None if not

Return type *VariableInfo*

class esmvalcore.cmor.table.**CMIP6Info**(*cmor_tables_path*, *default=None*, *alt_names=None*, *strict=True*, *default_table_prefix=''*)

Bases: *esmvalcore.cmor.table.InfoBase*

Class to read CMIP6-like data request.

This uses CMOR 3 json format

Parameters

- **cmor_tables_path** (*str*) – Path to the folder containing the Tables folder with the json files
- **default** (*object*) – Default table to look variables on if not found
- **strict** (*bool*) – If False, will look for a variable in other tables if it can not be found in the requested one

Methods:

get_table(table)	Search and return the table info.
get_variable(table_name, short_name[, derived])	Search and return the variable info.

get_table(*table*)

Search and return the table info.

Parameters **table** (*str*) – Table name

Returns Return the TableInfo object for the requested table if found, returns None if not

Return type *TableInfo*

get_variable(*table_name*, *short_name*, *derived=False*)

Search and return the variable info.

Parameters

- **table_name** (*str*) – Table name
- **short_name** (*str*) – Variable's short name
- **derived** (*bool, optional*) – Variable is derived. Info retrieval for derived variables always look on the default tables if variable is not find in the requested table

Returns Return the VariableInfo object for the requested variable if found, returns None if not

Return type *VariableInfo*

esmvalcore.cmor.table.**CMOR_TABLES: Dict[str, Type[*esmvalcore.cmor.table.InfoBase*]] = {'CMIP3': <esmvalcore.cmor.table.CMIP3Info object>, 'CMIP5': <esmvalcore.cmor.table.CMIP5Info object>, 'CMIP6': <esmvalcore.cmor.table.CMIP6Info object>, 'CORDEX': <esmvalcore.cmor.table.CMIP5Info object>, 'EMAC': <esmvalcore.cmor.table.CMIP5Info object>, 'IPSLCM': <esmvalcore.cmor.table.CMIP6Info object>, 'OBS': <esmvalcore.cmor.table.CMIP5Info object>, 'OBS6': <esmvalcore.cmor.table.CMIP6Info object>, 'ana4mips': <esmvalcore.cmor.table.CMIP5Info object>, 'custom': <esmvalcore.cmor.table.CustomInfo object>, 'native6': <esmvalcore.cmor.table.CMIP6Info object>, 'obs4mips': <esmvalcore.cmor.table.CMIP6Info object>}**

CMOR info objects.

Type dict of str, obj

class esmvalcore.cmor.table.**CoordinateInfo**(*name*)

Bases: *esmvalcore.cmor.table.JsonInfo*

Class to read and store coordinate information.

Attributes:

axis	Axis
generic_lev_name	Generic level name
long_name	Long name
must_have_bounds	Whether bounds are required on this dimension
out_name	Out name
requested	Values requested
standard_name	Standard name
stored_direction	Direction in which the coordinate increases
units	Units
valid_max	Maximum allowed value
valid_min	Minimum allowed value
value	Coordinate value
var_name	Short name

Methods:

read_json(json_data)	Read coordinate information from json.

axis

Axis

generic_lev_name

Generic level name

long_name

Long name

must_have_bounds

Whether bounds are required on this dimension

out_name

Out name

This is the name of the variable in the file

read_json(*json_data*)
Read coordinate information from json.

Non-present options will be set to empty

Parameters **json_data** (*dict*) – dictionary created by the json reader containing coordinate information

requested
Values requested

standard_name
Standard name

stored_direction
Direction in which the coordinate increases

units
Units

valid_max
Maximum allowed value

valid_min
Minimum allowed value

value
Coordinate value

var_name
Short name

class esmvalcore.cmor.table.**CustomInfo**(*cmor_tables_path=None*)
Bases: *esmvalcore.cmor.table.CMIP5Info*

Class to read custom var info for ESMVal.

Parameters **cmor_tables_path** (*str or None*) – Full path to the table or name for the table if it is present in ESMValTool repository

Methods:

get_table(table)	Search and return the table info.
get_variable(table, short_name[, derived])	Search and return the variable info.

get_table(*table*)
Search and return the table info.

Parameters **table** (*str*) – Table name

Returns Return the TableInfo object for the requested table if found, returns None if not

Return type *TableInfo*

get_variable(*table*, *short_name*, *derived=False*)
Search and return the variable info.

Parameters

- **table** (*str*) – Table name
- **short_name** (*str*) – Variable's short name

- **derived** (*bool, optional*) – Variable is derived. Info retrieval for derived variables always look on the default tables if variable is not find in the requested table

Returns Return the VariableInfo object for the requested variable if found, returns None if not

Return type *VariableInfo*

class esmvalcore.cmor.table.**InfoBase**(*default*, *alt_names*, *strict*)

Bases: object

Base class for all table info classes.

This uses CMOR 3 json format

Parameters

- **default** (*object*) – Default table to look variables on if not found
- **alt_names** (*list[list[str]]*) – List of known alternative names for variables
- **strict** (*bool*) – If False, will look for a variable in other tables if it can not be found in the requested one

Methods:

get_table(table)	Search and return the table info.
get_variable(table_name, short_name[, derived])	Search and return the variable info.

get_table(*table*)

Search and return the table info.

Parameters **table** (*str*) – Table name

Returns Return the TableInfo object for the requested table if found, returns None if not

Return type *TableInfo*

get_variable(*table_name*, *short_name*, *derived=False*)

Search and return the variable info.

Parameters

- **table_name** (*str*) – Table name
- **short_name** (*str*) – Variable's short name
- **derived** (*bool, optional*) – Variable is derived. Info retrieval for derived variables always look on the default tables if variable is not find in the requested table

Returns Return the VariableInfo object for the requested variable if found, returns None if not

Return type *VariableInfo*

class esmvalcore.cmor.table.**JsonInfo**

Bases: object

Base class for the info classes.

Provides common utility methods to read json variables

class esmvalcore.cmor.table.**TableInfo**(**args*, ***kwargs*)

Bases: dict

Container class for storing a CMOR table.

Methods:

clear()	
copy()	
fromkeys([value])	Create a new dictionary with keys from iterable and values set to value.
get(key[, default])	Return the value for key if key is in the dictionary, else default.
items()	
keys()	
pop(k[,d])	If key is not found, d is returned if given, otherwise KeyError is raised
popitem()	2-tuple; but raise KeyError if D is empty.
setdefault(key[, default])	Insert key with a value of default if key is not in the dictionary.
update([E,]**F)	If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]
values()	

clear() → None. Remove all items from D.

copy() → a shallow copy of D

fromkeys(*value=None*, */*)
Create a new dictionary with keys from iterable and values set to value.

get(*key*, *default=None*, */*)
Return the value for key if key is in the dictionary, else default.

items() → a set-like object providing a view on D's items

keys() → a set-like object providing a view on D's keys

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised

popitem() → (k, v), remove and return some (key, value) pair as a
2-tuple; but raise KeyError if D is empty.

setdefault(*key*, *default=None*, */*)
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

update([*E*], ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

class esmvalcore.cmor.table.**VariableInfo**(*table_type*, *short_name*)
Bases: *esmvalcore.cmor.table.JsonInfo*

Class to read and store variable information.

Attributes:

coordinates	Coordinates
dimensions	List of dimensions
frequency	Data frequency
long_name	Long name
modeling_realm	Modeling realm
positive	Increasing direction
short_name	Short name
standard_name	Standard name
units	Data units
valid_max	Maximum admitted value
valid_min	Minimum admitted value

Methods:

copy()	Return a shallow copy of VariableInfo.
read_json(json_data, default_freq)	Read variable information from json.

coordinates
Coordinates

This is a dict with the names of the dimensions as keys and CoordinateInfo objects as values.

copy()
Return a shallow copy of VariableInfo.

Returns Shallow copy of this object

Return type *VariableInfo*

dimensions
List of dimensions

frequency
Data frequency

long_name
Long name

modeling_realm
Modeling realm

positive
Increasing direction

read_json(*json_data*, *default_freq*)
Read variable information from json.

Non-present options will be set to empty

Parameters

- **json_data** (*dict*) – dictionary created by the json reader containing variable information
- **default_freq** (*str*) – Default frequency to use if it is not defined at variable level

short_name
Short name

standard_name
Standard name

units
Data units

valid_max
Maximum admitted value

valid_min
Minimum admitted value

esmvalcore.cmor.table.**get_var_info**(*project*, *mip*, *short_name*)
Get variable information.

Parameters

- **project** (*str*) – Dataset's project.
- **mip** (*str*) – Variable's cmor table.
- **short_name** (*str*) – Variable's short name.

esmvalcore.cmor.table.**read_cmor_tables**(*cfg_developer=None*)
Read cmor tables required in the configuration.

Parameters **cfg_developer** (*dict of str*) – Parsed config-developer file

CHAPTER

TWENTYEIGHT

PREPROCESSOR FUNCTIONS

By default, preprocessor functions are applied in the order in which they are listed here.

Preprocessor module.

Functions:

add_fx_variables(cube, fx_variables, check_level)	Load requested fx files, check with CMOR standards and add the fx variables as cell measures or ancillary variables in the cube containing the data.
amplitude(cube, coords)	Calculate amplitude of cycles by aggregating over coordinates.
annual_statistics(cube[, operator])	Compute annual statistics.
anomalies(cube, period[, reference, ...])	Compute anomalies using a mean with the specified granularity.
area_statistics(cube, operator)	Apply a statistical operator in the horizontal direction.
cleanup(files[, remove])	Clean up after running the preprocessor.
climate_statistics(cube[, operator, period, ...])	Compute climate statistics with the specified granularity.
clip(cube[, minimum, maximum])	Clip values at a specified minimum and/or maximum value
clip_start_end_year(cube, start_year, end_year)	Extract time range given by the dataset keys.
cmor_check_data(cube, cmor_table, mip, ...)	Check if data conforms to variable's CMOR definition.
cmor_check_metadata(cube, cmor_table, mip, ...)	Check if metadata conforms to variable's CMOR definition.
concatenate(cubes)	Concatenate all cubes after fixing metadata.
convert_units(cube, units)	Convert the units of a cube to new ones.
daily_statistics(cube[, operator])	Compute daily statistics.
decadal_statistics(cube[, operator])	Compute decadal statistics.
depth_integration(cube)	Determine the total sum over the vertical component.
derive(cubes, short_name, long_name, units)	Derive variable.
detrend(cube[, dimension, method])	Detrend data along a given dimension.
download(files, dest_folder)	Download files that are not available locally.
extract_levels(cube, levels, scheme[, ...])	Perform vertical interpolation.
extract_month(cube, month)	Slice cube to get only the data belonging to a specific month.
extract_named_regions(cube, regions)	Extract a specific named region.
extract_point(cube, latitude, longitude, scheme)	Extract a point, with interpolation.
extract_region(cube, start_longitude, ...)	Extract a region from a cube.
extract_season(cube, season)	Slice cube to get only the data belonging to a specific season.
extract_shape(cube, shapefile[, method, ...])	Extract a region defined by a shapefile.

continues on next page

Table 1 – continued from previous page

extract_time(cube, start_year, start_month, ...)	Extract a time range from a cube.
extract_trajectory(cube, latitudes, longitudes)	Extract data along a trajectory.
extract_transect(cube[, latitude, longitude])	Extract data along a line of constant latitude or longitude.
extract_volume(cube, z_min, z_max)	Subset a cube based on a range of values in the z-coordinate.
fix_data(cube, short_name, project, dataset, mip)	Fix cube data if fixes add present and check it anyway.
fix_file(file, short_name, project, dataset, ...)	Fix files before ESMValTool can load them.
fix_metadata(cubes, short_name, project, ...)	Fix cube metadata if fixes are required and check it anyway.
hourly_statistics(cube, hours[, operator])	Compute hourly statistics.
linear_trend(cube[, coordinate])	Calculate linear trend of data along a given coordinate.
linear_trend_stderr(cube[, coordinate])	Calculate standard error of linear trend along a given coordinate.
load(file[, callback])	Load iris cubes from files.
mask_above_threshold(cube, threshold)	Mask above a specific threshold value.
mask_below_threshold(cube, threshold)	Mask below a specific threshold value.
mask_fillvalues(products, threshold_fraction)	Compute and apply a multi-dataset fillvalues mask.
mask_glaciated(cube, mask_out)	Mask out glaciated areas.
mask_inside_range(cube, minimum, maximum)	Mask inside a specific threshold range.
mask_landsea(cube, mask_out[, ...])	Mask out either land mass or sea (oceans, seas and lakes).
mask_landseaice(cube, mask_out)	Mask out either landsea (combined) or ice.
mask_multimodel(products)	Apply common mask to all datasets (using logical OR).
mask_outside_range(cube, minimum, maximum)	Mask outside a specific threshold range.
meridional_statistics(cube, operator)	Compute meridional statistics.
monthly_statistics(cube[, operator])	Compute monthly statistics.
multi_model_statistics(products, span, ...)	Compute multi-model statistics.
regrid(cube, target_grid, scheme[, ...])	Perform horizontal regridding.
regrid_time(cube, frequency)	Align time axis for cubes so they can be subtracted.
remove_fx_variables(cube)	Remove fx variables present as cell measures or ancillary variables in the cube containing the data.
resample_hours(cube, interval[, offset])	Convert x-hourly data to y-hourly by eliminating extra timesteps.
resample_time(cube[, month, day, hour])	Change frequency of data by resampling it.
save(cubes, filename[, optimize_access, ...])	Save iris cubes to file.
seasonal_statistics(cube[, operator, seasons])	Compute seasonal statistics.
timeseries_filter(cube, window, span[, ...])	Apply a timeseries filter.
volume_statistics(cube, operator)	Apply a statistical operation over a volume.
weighting_landsea_fraction(cube, area_type)	Weight fields using land or sea fraction.
zonal_statistics(cube, operator)	Compute zonal statistics.

esmvalcore.preprocessor.**add_fx_variables**(*cube*, *fx_variables*, *check_level*)

Load requested fx files, check with CMOR standards and add the fx variables as cell measures or ancillary variables in the cube containing the data.

Parameters

- **cube** (*iris.cube.Cube*) – Iris cube with input data.
- **fx_variables** (*dict*) – Dictionary with fx_variable information.
- **check_level** (CheckLevels) – Level of strictness of the checks.

Returns Cube with added cell measures or ancillary variables.

Return type iris.cube.Cube

esmvalcore.preprocessor.**amplitude**(*cube*, *coords*)

Calculate amplitude of cycles by aggregating over coordinates.

Note: The amplitude is calculated as *peak-to-peak* amplitude (difference between maximum and minimum value of the signal). Other amplitude types are currently not supported.

Parameters

- **cube** (*iris.cube.Cube*) – Input data.
- **coords** (*str or list of str*) – Coordinates over which is aggregated. For example, use `'year'` to extract the annual cycle amplitude for each year in the data or `['day_of_year', 'year']` to extract the diurnal cycle amplitude for each individual day in the data. If the coordinates are not found in `cube`, try to add it via `iris.coord_categorisation` (at the moment, this only works for the temporal coordinates `day_of_month`, `day_of_year`, `hour`, `month`, `month_fullname`, `month_number`, `season`, `season_number`, `season_year`, `weekday`, `weekday_fullname`, `weekday_number` or `year`.

Returns Amplitudes.

Return type iris.cube.Cube

Raises **iris.exceptions.CoordinateNotFoundError** – A coordinate is not found in `cube` and cannot be added via `iris.coord_categorisation`.

esmvalcore.preprocessor.**annual_statistics**(*cube*, *operator='mean'*)

Compute annual statistics.

Note that this function does not weight the annual mean if uneven time periods are present. Ie, all data inside the year are treated equally.

Parameters

- **cube** (*iris.cube.Cube*) – input cube.
- **operator** (*str, optional*) – Select operator to apply. Available operators: 'mean', 'median', 'std_dev', 'sum', 'min', 'max', 'rms'

Returns Annual statistics cube

Return type iris.cube.Cube

esmvalcore.preprocessor.**anomalies**(*cube*, *period*, *reference=None*, *standardize=False*, *seasons=('DJF', 'MAM', 'JJA', 'SON')*)

Compute anomalies using a mean with the specified granularity.

Computes anomalies based on daily, monthly, seasonal or yearly means for the full available period

Parameters

- **cube** (*iris.cube.Cube*) – input cube.
- **period** (*str*) – Period to compute the statistic over. Available periods: 'full', 'season', 'seasonal', 'monthly', 'month', 'mon', 'daily', 'day'
- **reference** (*list int, optional, default: None*) – Period of time to use a reference, as needed for the 'extract_time' preprocessor function If None, all available data is used as a reference

- **standardize** (*bool, optional*) – If True standardized anomalies are calculated
- **seasons** (*list or tuple of str, optional*) – Seasons to use if needed. Defaults to ('DJF', 'MAM', 'JJA', 'SON')

Returns Anomalies cube

Return type iris.cube.Cube

esmvalcore.preprocessor.**area_statistics**(*cube*, *operator*)

Apply a statistical operator in the horizontal direction.

The average in the horizontal direction. We assume that the horizontal directions are ['longitude', 'latutude'].

This function can be used to apply several different operations in the horizontal plane: mean, standard deviation, median variance, minimum and maximum. These options are specified using the *operator* argument and the following key word arguments:

mean	Area weighted mean.
median	Median (not area weighted)
std_dev	Standard Deviation (not area weighted)
sum	Area weighted sum.
variance	Variance (not area weighted)
min:	Minimum value
max	Maximum value
rms	Area weighted root mean square.

Parameters

- **cube** (*iris.cube.Cube*) – Input cube.
- **operator** (*str*) – The operation, options: mean, median, min, max, std_dev, sum, variance, rms.

Returns collapsed cube.

Return type iris.cube.Cube

Raises

- **iris.exceptions.CoordinateMultiDimError** – Exception for latitude axis with dim > 2.
- **ValueError** – if input data cube has different shape than grid area weights

esmvalcore.preprocessor.**cleanup**(*files*, *remove=None*)

Clean up after running the preprocessor.

esmvalcore.preprocessor.**climate_statistics**(*cube*, *operator='mean'*, *period='full'*, *seasons=('DJF', 'MAM', 'JJA', 'SON')*)

Compute climate statistics with the specified granularity.

Computes statistics for the whole dataset. It is possible to get them for the full period or with the data grouped by day, month or season

Parameters

- **cube** (*iris.cube.Cube*) – input cube.
- **operator** (*str, optional*) – Select operator to apply. Available operators: 'mean', 'median', 'std_dev', 'sum', 'min', 'max', 'rms'

- **period** (*str, optional*) – Period to compute the statistic over. Available periods: 'full', 'season', 'seasonal', 'monthly', 'month', 'mon', 'daily', 'day'
- **seasons** (*list or tuple of str, optional*) – Seasons to use if needed. Defaults to ('DJF', 'MAM', 'JJA', 'SON')

Returns Monthly statistics cube

Return type iris.cube.Cube

esmvalcore.preprocessor.**clip**(*cube*, *minimum=None*, *maximum=None*)

Clip values at a specified minimum and/or maximum value

Values lower than minimum are set to minimum and values higher than maximum are set to maximum.

Parameters

- **cube** (*iris.cube.Cube*) – iris cube to be clipped
- **minimum** (*float*) – lower threshold to be applied on input cube data.
- **maximum** (*float*) – upper threshold to be applied on input cube data.

Returns clipped cube.

Return type iris.cube.Cube

esmvalcore.preprocessor.**clip_start_end_year**(*cube*, *start_year*, *end_year*)

Extract time range given by the dataset keys.

Parameters

- **cube** (*iris.cube.Cube*) – Input cube.
- **start_year** (*int*) – Start year.
- **end_year** (*int*) – End year.

Returns Sliced cube.

Return type iris.cube.Cube

Raises **ValueError** – Time ranges are outside the cube's time limits.

esmvalcore.preprocessor.**cmor_check_data**(*cube*, *cmor_table*, *mip*, *short_name*, *frequency*, *check_level=<CheckLevels.DEFAULT: 3>*)

Check if data conforms to variable's CMOR definition.

The checks performed at this step require the data in memory.

Parameters

- **cube** (*iris.cube.Cube*) – Data cube to check.
- **cmor_table** (*str*) – CMOR definitions to use.
- **mip** – Variable's mip.
- **short_name** (*str*) – Variable's short name
- **frequency** (*str*) – Data frequency
- **check_level** (CheckLevels) – Level of strictness of the checks.

esmvalcore.preprocessor.**cmor_check_metadata**(*cube*, *cmor_table*, *mip*, *short_name*, *frequency*, *check_level=<CheckLevels.DEFAULT: 3>*)

Check if metadata conforms to variable's CMOR definition.

None of the checks at this step will force the cube to load the data.

Parameters

- **cube** (*iris.cube.Cube*) – Data cube to check.
- **cmor_table** (*str*) – CMOR definitions to use.
- **mip** – Variable's mip.
- **short_name** (*str*) – Variable's short name.
- **frequency** (*str*) – Data frequency.
- **check_level** (CheckLevels) – Level of strictness of the checks.

esmvalcore.preprocessor.**concatenate**(*cubes*)
Concatenate all cubes after fixing metadata.

esmvalcore.preprocessor.**convert_units**(*cube*, *units*)
Convert the units of a cube to new ones.

This converts units of a cube.

Parameters

- **cube** (*iris.cube.Cube*) – input cube
- **units** (*str*) – new units in udunits form

Returns converted cube.

Return type iris.cube.Cube

esmvalcore.preprocessor.**daily_statistics**(*cube*, *operator='mean'*)
Compute daily statistics.

Chunks time in daily periods and computes statistics over them;

Parameters

- **cube** (*iris.cube.Cube*) – input cube.
- **operator** (*str, optional*) – Select operator to apply. Available operators: 'mean', 'median', 'std_dev', 'sum', 'min', 'max', 'rms'

Returns Daily statistics cube

Return type iris.cube.Cube

esmvalcore.preprocessor.**decadal_statistics**(*cube*, *operator='mean'*)
Compute decadal statistics.

Note that this function does not weight the decadal mean if uneven time periods are present. Ie, all data inside the decade are treated equally.

Parameters

- **cube** (*iris.cube.Cube*) – input cube.
- **operator** (*str, optional*) – Select operator to apply. Available operators: 'mean', 'median', 'std_dev', 'sum', 'min', 'max', 'rms'

Returns Decadal statistics cube

Return type iris.cube.Cube

esmvalcore.preprocessor.**depth_integration**(*cube*)
Determine the total sum over the vertical component.

Requires a 3D cube. The z-coordinate integration is calculated by taking the sum in the z direction of the cell contents multiplied by the cell thickness.

Parameters **cube** (*iris.cube.Cube*) – input cube.

Returns collapsed cube.

Return type iris.cube.Cube

esmvalcore.preprocessor.**derive**(*cubes*, *short_name*, *long_name*, *units*, *standard_name=None*)
Derive variable.

Parameters

- **cubes** (*iris.cube.CubeList*) – Includes all the needed variables for derivation defined in `get_required()`.
- **short_name** (*str*) – short_name
- **long_name** (*str*) – long_name
- **units** (*str*) – units
- **standard_name** (*str, optional*) – standard_name

Returns The new derived variable.

Return type iris.cube.Cube

esmvalcore.preprocessor.**detrend**(*cube*, *dimension='time'*, *method='linear'*)
Detrend data along a given dimension.

Parameters

- **cube** (*iris.cube.Cube*) – input cube.
- **dimension** (*str*) – Dimension to detrend
- **method** (*str*) – Method to detrend. Available: linear, constant. See documentation of 'scipy.signal.detrend' for details

Returns Detrended cube

Return type iris.cube.Cube

esmvalcore.preprocessor.**download**(*files*, *dest_folder*)
Download files that are not available locally.

esmvalcore.preprocessor.**extract_levels**(*cube*, *levels*, *scheme*, *coordinate=None*)
Perform vertical interpolation.

Parameters

- **cube** (*cube*) – The source cube to be vertically interpolated.
- **levels** (*array*) – One or more target levels for the vertical interpolation. Assumed to be in the same S.I. units of the source cube vertical dimension coordinate.
- **scheme** (*str*) – The vertical interpolation scheme to use. Choose from 'linear', 'nearest', 'nearest_horizontal_extrapolate_vertical', 'linear_horizontal_extrapolate_vertical'.
- **coordinate** (*optional str*) – The coordinate to interpolate. If specified, pressure levels (if present) can be converted to height levels and vice versa using the US standard atmosphere. E.g. 'coordinate = altitude' will convert existing pressure levels (air_pressure) to height levels (altitude); 'coordinate = air_pressure' will convert existing height levels (altitude) to pressure levels (air_pressure).

Returns

Return type cube

See also:

regrid Perform horizontal regridding.

esmvalcore.preprocessor.**extract_month**(*cube*, *month*)
Slice cube to get only the data belonging to a specific month.

Parameters

- **cube** (*iris.cube.Cube*) – Original data
- **month** (*int*) – Month to extract as a number from 1 to 12

Returns data cube for specified month.

Return type iris.cube.Cube

Raises **ValueError** – if requested month is not present in the cube

esmvalcore.preprocessor.**extract_named_regions**(*cube*, *regions*)
Extract a specific named region.

The region coordinate exist in certain CMIP datasets. This preprocessor allows a specific named regions to be extracted.

Parameters

- **cube** (*iris.cube.Cube*) – input cube.
- **regions** (*str, list*) – A region or list of regions to extract.

Returns collapsed cube.

Return type iris.cube.Cube

Raises

- **ValueError** – regions is not list or tuple or set.
- **ValueError** – region not included in cube.

esmvalcore.preprocessor.**extract_point**(*cube*, *latitude*, *longitude*, *scheme*)
Extract a point, with interpolation.

Extracts a single latitude/longitude point from a cube, according to the interpolation scheme *scheme*.

Multiple points can also be extracted, by supplying an array of latitude and/or longitude coordinates. The resulting point cube will match the respective latitude and longitude coordinate to those of the input coordinates. If the input coordinate is a scalar, the dimension will be missing in the output cube (that is, it will be a scalar).

Parameters

- **cube** (*cube*) – The source cube to extract a point from.
- **latitude** (*float, or array of float*) – The latitude and longitude of the point.
- **longitude** (*float, or array of float*) – The latitude and longitude of the point.
- **scheme** (*str*) – The interpolation scheme. 'linear' or 'nearest'. No default.

Returns

- *Returns a cube with the extracted point(s), and with adjusted*

- *latitude and longitude coordinates (see above).*

Examples

With a cube that has the coordinates

- latitude: [1, 2, 3, 4]
- longitude: [1, 2, 3, 4]
- **data values: [[[1, 2, 3, 4], [5, 6, ...], [...], [...], ...]]**

```
>>> point = extract_point(cube, 2.5, 2.5, 'linear')
>>> point.data
array([ 8.5, 24.5, 40.5, 56.5])
```

Extraction of multiple points at once, with a nearest matching scheme. The values for 0.1 will result in masked values, since this lies outside the cube grid.

```
>>> point = extract_point(cube, [1.4, 2.1], [0.1, 1.1],
...                       'nearest')
>>> point.data.shape
(4, 2, 2)
>>> # x, y, z indices of masked values
>>> np.where(~point.data.mask)
(array([0, 0, 1, 1, 2, 2, 3, 3]), array([0, 1, 0, 1, 0, 1, 0, 1]),
array([1, 1, 1, 1, 1, 1, 1, 1]))
>>> point.data[~point.data.mask].data
array([ 1,  5, 17, 21, 33, 37, 49, 53])
```

esmvalcore.preprocessor.**extract_region**(*cube*, *start_longitude*, *end_longitude*, *start_latitude*, *end_latitude*)

Extract a region from a cube.

Function that subsets a cube on a box (start_longitude, end_longitude, start_latitude, end_latitude)

Parameters

- **cube** (*iris.cube.Cube*) – input data cube.
- **start_longitude** (*float*) – Western boundary longitude.
- **end_longitude** (*float*) – Eastern boundary longitude.
- **start_latitude** (*float*) – Southern Boundary latitude.
- **end_latitude** (*float*) – Northern Boundary Latitude.

Returns smaller cube.

Return type iris.cube.Cube

esmvalcore.preprocessor.**extract_season**(*cube*, *season*)

Slice cube to get only the data belonging to a specific season.

Parameters

- **cube** (*iris.cube.Cube*) – Original data
- **season** (*str*) – Season to extract. Available: DJF, MAM, JJA, SON and all sequentially correct combinations: e.g. JJAS

Returns data cube for specified season.

Return type iris.cube.Cube

Raises `ValueError` – if requested season is not present in the cube

esmvalcore.preprocessor.**extract_shape**(*cube*, *shapefile*, *method='contains'*, *crop=True*, *decomposed=False*, *ids=None*)

Extract a region defined by a shapefile.

Note that this function does not work for shapes crossing the prime meridian or poles.

Parameters

- **cube** (*iris.cube.Cube*) – input cube.
- **shapefile** (*str*) – A shapefile defining the region(s) to extract.
- **method** (*str, optional*) – Select all points contained by the shape or select a single representative point. Choose either 'contains' or 'representative'. If 'contains' is used, but not a single grid point is contained by the shape, a representative point will selected.
- **crop** (*bool, optional*) – Crop the resulting cube using *extract_region()*. Note that data on irregular grids will not be cropped.
- **decomposed** (*bool, optional*) – Whether or not to retain the sub shapes of the shapefile in the output. If this is set to True, the output cube has a dimension for the sub shapes.
- **ids** (*list(str), optional*) – List of shapes to be read from the file. The ids are assigned from the attributes 'name' or 'id' (in that priority order) if present in the file or correspond to the reading order if not.

Returns Cube containing the extracted region.

Return type iris.cube.Cube

See also:

extract_region Extract a region from a cube.

esmvalcore.preprocessor.**extract_time**(*cube*, *start_year*, *start_month*, *start_day*, *end_year*, *end_month*, *end_day*)

Extract a time range from a cube.

Given a time range passed in as a series of years, months and days, it returns a time-extracted cube with data only within the specified time range.

Parameters

- **cube** (*iris.cube.Cube*) – input cube.
- **start_year** (*int*) – start year
- **start_month** (*int*) – start month
- **start_day** (*int*) – start day
- **end_year** (*int*) – end year
- **end_month** (*int*) – end month
- **end_day** (*int*) – end day

Returns Sliced cube.

Return type iris.cube.Cube

Raises ValueError – if time ranges are outside the cube time limits

esmvalcore.preprocessor.**extract_trajectory**(*cube*, *latitudes*, *longitudes*, *number_points=2*)

Extract data along a trajectory.

latitudes and longitudes are the pairs of coordinates for two points. number_points is the number of points between the two points.

This version uses the expensive interpolate method, but it may be necceasiry for irregular grids.

If only two latitude and longitude coordinates are given, extract_trajectory will produce a cube will extrapolate along a line between those two points, and will add *number_points* points between the two corners.

If more than two points are provided, then extract_trajectory will produce a cube which has extrapolated the data of the cube to those points, and *number_points* is not needed.

Parameters

- **cube** (*iris.cube.Cube*) – input cube.
- **latitudes** (*list*) – list of latitude coordinates (floats).
- **longitudes** (*list*) – list of longitude coordinates (floats).
- **number_points** (*int*) – number of points to extrapolate (optional).

Returns collapsed cube.

Return type iris.cube.Cube

Raises ValueError – if latitude and longitude have different dimensions.

esmvalcore.preprocessor.**extract_transect**(*cube*, *latitude=None*, *longitude=None*)

Extract data along a line of constant latitude or longitude.

Both arguments, latitude and longitude, are treated identically. Either argument can be a single float, or a pair of floats, or can be left empty. The single float indicates the latitude or longitude along which the transect should be extracted. A pair of floats indicate the range that the transect should be extracted along the secondairy axis.

For instance *'extract_transect(cube, longitude=-28)'* will produce a transect along 28 West.

Also, *'extract_transect(cube, longitude=-28, latitude=[-50, 50])'* will produce a transect along 28 West between 50 south and 50 North.

This function is not yet implemented for irregular arrays - instead try the extract_trajectory function, but note that it is currently very slow. Alternatively, use the regrid preprocessor to regrid along a regular grid and then extract the transect.

Parameters

- **cube** (*iris.cube.Cube*) – input cube.
- **latitude** (*None, float or [float, float], optional*) – transect latiude or range.
- **longitude** (*None, float or [float, float], optional*) – transect longitude or range.

Returns collapsed cube.

Return type iris.cube.Cube

Raises

- ValueError – slice extraction not implemented for irregular grids.
- ValueError – latitude and longitude are both floats or lists; not allowed to slice on both axes at the same time.

esmvalcore.preprocessor.**extract_volume**(*cube*, *z_min*, *z_max*)

Subset a cube based on a range of values in the z-coordinate.

Function that subsets a cube on a box (z_min, z_max) This function is a restriction of masked_cube_lonlat(); Note that this requires the requested z-coordinate range to be the same sign as the iris cube. ie, if the cube has z-coordinate as negative, then z_min and z_max need to be negative numbers.

Parameters

- **cube** (*iris.cube.Cube*) – input cube.
- **z_min** (*float*) – minimum depth to extract.
- **z_max** (*float*) – maximum depth to extract.

Returns z-coord extracted cube.

Return type iris.cube.Cube

esmvalcore.preprocessor.**fix_data**(*cube*, *short_name*, *project*, *dataset*, *mip*, *frequency=None*, *check_level=<CheckLevels.DEFAULT: 3>*, ***extra_facets*)

Fix cube data if fixes add present and check it anyway.

This method assumes that metadata is already fixed and checked.

This method collects all the relevant fixes for a given variable, applies them and checks resulting cube (or the original if no fixes were needed) metadata to ensure that it complies with the standards of its project CMOR tables.

Parameters

- **cube** (*iris.cube.Cube*) – Cube to fix
- **short_name** (*str*) – Variable's short name
- **project** (*str*) –
- **dataset** (*str*) –
- **mip** (*str*) – Variable's MIP
- **frequency** (*str, optional*) – Variable's data frequency, if available
- **check_level** (CheckLevels) – Level of strictness of the checks. Set to default.
- ****extra_facets** (*dict, optional*) – Extra facets are mainly used for data outside of the big projects like CMIP, CORDEX, obs4MIPs. For details, see *Extra Facets*.

Returns Fixed and checked cube

Return type iris.cube.Cube

Raises ***CMORCheckError*** – If the checker detects errors in the data that it can not fix.

esmvalcore.preprocessor.**fix_file**(*file*, *short_name*, *project*, *dataset*, *mip*, *output_dir*, ***extra_facets*)

Fix files before ESMValTool can load them.

This fixes are only for issues that prevent iris from loading the cube or that cannot be fixed after the cube is loaded.

Original files are not overwritten.

Parameters

- **file** (*str*) – Path to the original file
- **short_name** (*str*) – Variable's short name

- **project** (*str*) –
- **dataset** (*str*) –
- **output_dir** (*str*) – Output directory for fixed files
- ****extra_facets** (*dict, optional*) – Extra facets are mainly used for data outside of the big projects like CMIP, CORDEX, obs4MIPs. For details, see *Extra Facets*.

Returns Path to the fixed file

Return type str

esmvalcore.preprocessor.**fix_metadata**(*cubes*, *short_name*, *project*, *dataset*, *mip*, *frequency=None*, *check_level=<CheckLevels.DEFAULT: 3>*, ***extra_facets*)

Fix cube metadata if fixes are required and check it anyway.

This method collects all the relevant fixes for a given variable, applies them and checks the resulting cube (or the original if no fixes were needed) metadata to ensure that it complies with the standards of its project CMOR tables.

Parameters

- **cubes** (*iris.cube.CubeList*) – Cubes to fix
- **short_name** (*str*) – Variable's short name
- **project** (*str*) –
- **dataset** (*str*) –
- **mip** (*str*) – Variable's MIP
- **frequency** (*str, optional*) – Variable's data frequency, if available
- **check_level** (CheckLevels) – Level of strictness of the checks. Set to default.
- ****extra_facets** (*dict, optional*) – Extra facets are mainly used for data outside of the big projects like CMIP, CORDEX, obs4MIPs. For details, see *Extra Facets*.

Returns Fixed and checked cube

Return type iris.cube.Cube

Raises ***CMORCheckError*** – If the checker detects errors in the metadata that it can not fix.

esmvalcore.preprocessor.**hourly_statistics**(*cube*, *hours*, *operator='mean'*)

Compute hourly statistics.

Chunks time in x hours periods and computes statistics over them.

Parameters

- **cube** (*iris.cube.Cube*) – input cube.
- **hours** (*int*) – Number of hours per period. Must be a divisor of 24 (1, 2, 3, 4, 6, 8, 12)
- **operator** (*str, optional*) – Select operator to apply. Available operators: 'mean', 'median', 'std_dev', 'sum', 'min', 'max'

Returns Hourly statistics cube

Return type iris.cube.Cube

esmvalcore.preprocessor.**linear_trend**(*cube*, *coordinate='time'*)

Calculate linear trend of data along a given coordinate.

The linear trend is defined as the slope of an ordinary linear regression.

Parameters

- **cube** (*iris.cube.Cube*) – Input data.
- **coordinate** (*str, optional (default: 'time')*) – Dimensional coordinate over which the trend is calculated.

Returns Trends.

Return type iris.cube.Cube

Raises **iris.exceptions.CoordinateNotFoundError** – The dimensional coordinate with the name `coordinate` is not found in `cube`.

esmvalcore.preprocessor.**linear_trend_stderr**(*cube*, *coordinate='time'*)

Calculate standard error of linear trend along a given coordinate.

This gives the standard error (not confidence intervals!) of the trend defined as the standard error of the estimated slope of an ordinary linear regression.

Parameters

- **cube** (*iris.cube.Cube*) – Input data.
- **coordinate** (*str, optional (default: 'time')*) – Dimensional coordinate over which the standard error of the trend is calculated.

Returns Standard errors of trends.

Return type iris.cube.Cube

Raises **iris.exceptions.CoordinateNotFoundError** – The dimensional coordinate with the name `coordinate` is not found in `cube`.

esmvalcore.preprocessor.**load**(*file*, *callback=None*)

Load iris cubes from files.

esmvalcore.preprocessor.**mask_above_threshold**(*cube*, *threshold*)

Mask above a specific threshold value.

Takes a value 'threshold' and masks off anything that is above it in the cube data. Values equal to the threshold are not masked.

Parameters

- **cube** (*iris.cube.Cube*) – iris cube to be thresholded.
- **threshold** (*float*) – threshold to be applied on input cube data.

Returns thresholded cube.

Return type iris.cube.Cube

esmvalcore.preprocessor.**mask_below_threshold**(*cube*, *threshold*)

Mask below a specific threshold value.

Takes a value 'threshold' and masks off anything that is below it in the cube data. Values equal to the threshold are not masked.

Parameters

- **cube** (*iris.cube.Cube*) – iris cube to be thresholded
- **threshold** (*float*) – threshold to be applied on input cube data.

Returns thresholded cube.

Return type iris.cube.Cube

esmvalcore.preprocessor.**mask_fillvalues**(*products*, *threshold_fraction*, *min_value=None*, *time_window=1*)

Compute and apply a multi-dataset fillvalues mask.

Construct the mask that fills a certain time window with missing values if the number of values in that specific window is less than a given fractional threshold. This function is the extension of _get_fillvalues_mask and performs the combination of missing values masks from each model (of multimodels) into a single fillvalues mask to be applied to each model.

Parameters

- **products** (*iris.cube.Cube*) – data products to be masked.
- **threshold_fraction** (*float*) – fractional threshold to be used as argument for Aggregator. Must be between 0 and 1.
- **min_value** (*float*) – minimum value threshold; default None If default, no thresholding applied so the full mask will be selected.
- **time_window** (*float*) – time window to compute missing data counts; default set to 1.

Returns Masked iris cubes.

Return type iris.cube.Cube

Raises **NotImplementedError** – Implementation missing for data with higher dimensionality than 4.

esmvalcore.preprocessor.**mask_glaciated**(*cube*, *mask_out*)

Mask out glaciated areas.

It applies a Natural Earth mask. Note that for computational reasons only the 10 largest polygons are used for masking.

Parameters

- **cube** (*iris.cube.Cube*) – data cube to be masked.
- **mask_out** (*str*) – "glaciated" to mask out glaciated areas

Returns Returns the masked iris cube.

Return type iris.cube.Cube

Raises **ValueError** – Error raised if masking on irregular grids is attempted or if mask_out has a wrong value.

esmvalcore.preprocessor.**mask_inside_range**(*cube*, *minimum*, *maximum*)

Mask inside a specific threshold range.

Takes a MINIMUM and a MAXIMUM value for the range, and masks off anything that's between the two in the cube data.

Parameters

- **cube** (*iris.cube.Cube*) – iris cube to be thresholded
- **minimum** (*float*) – lower threshold to be applied on input cube data.
- **maximum** (*float*) – upper threshold to be applied on input cube data.

Returns thresholded cube.

Return type iris.cube.Cube

esmvalcore.preprocessor.**mask_landsea**(*cube*, *mask_out*, *always_use_ne_mask=False*)
Mask out either land mass or sea (oceans, seas and lakes).

It uses dedicated ancillary variables (sftlf or sftof) or, in their absence, it applies a Natural Earth mask (land or ocean contours). Note that the Natural Earth masks have different resolutions: 10m for land, and 50m for seas. These are more than enough for ESMValTool purposes.

Parameters

- **cube** (*iris.cube.Cube*) – data cube to be masked.
- **mask_out** (*str*) – either "land" to mask out land mass or "sea" to mask out seas.
- **always_use_ne_mask** (*bool, optional (default: False)*) – always apply Natural Earths mask, regardless if fx files are available or not.

Returns Returns the masked iris cube.

Return type iris.cube.Cube

Raises **ValueError** – Error raised if masking on irregular grids is attempted. Irregular grids are not currently supported for masking with Natural Earth shapefile masks.

esmvalcore.preprocessor.**mask_landseaice**(*cube*, *mask_out*)
Mask out either landsea (combined) or ice.

Function that masks out either landsea (land and seas) or ice (Antarctica and Greenland and some wee glaciers).

It uses dedicated ancillary variables (sftgif).

Parameters

- **cube** (*iris.cube.Cube*) – data cube to be masked.
- **mask_out** (*str*) – either "landsea" to mask out landsea or "ice" to mask out ice.

Returns Returns the masked iris cube with either land or ice masked out.

Return type iris.cube.Cube

Raises **ValueError** – Error raised if landsea-ice mask not found as an ancillary variable.

esmvalcore.preprocessor.**mask_multimodel**(*products*)
Apply common mask to all datasets (using logical OR).

Parameters **products** (*iris.cube.CubeList or list of PreprocessorFile*) – Data products/cubes to be masked.

Returns Masked data products/cubes.

Return type iris.cube.CubeList or list of PreprocessorFile

Raises

- **ValueError** – Datasets have different shapes.
- **TypeError** – Invalid input data.

esmvalcore.preprocessor.**mask_outside_range**(*cube*, *minimum*, *maximum*)
Mask outside a specific threshold range.

Takes a MINIMUM and a MAXIMUM value for the range, and masks off anything that's outside the two in the cube data.

Parameters

- **cube** (*iris.cube.Cube*) – iris cube to be thresholded

- **minimum** (*float*) – lower threshold to be applied on input cube data.
- **maximum** (*float*) – upper threshold to be applied on input cube data.

Returns thresholded cube.

Return type iris.cube.Cube

esmvalcore.preprocessor.**meridional_statistics**(*cube*, *operator*)

Compute meridional statistics.

Parameters

- **cube** (*iris.cube.Cube*) – input cube.
- **operator** (*str, optional*) – Select operator to apply. Available operators: 'mean', 'median', 'std_dev', 'sum', 'min', 'max', 'rms'.

Returns Meridional statistics cube.

Return type iris.cube.Cube

Raises **ValueError** – Error raised if computation on irregular grids is attempted. Zonal statistics not yet implemented for irregular grids.

esmvalcore.preprocessor.**monthly_statistics**(*cube*, *operator='mean'*)

Compute monthly statistics.

Chunks time in monthly periods and computes statistics over them;

Parameters

- **cube** (*iris.cube.Cube*) – input cube.
- **operator** (*str, optional*) – Select operator to apply. Available operators: 'mean', 'median', 'std_dev', 'sum', 'min', 'max', 'rms'

Returns Monthly statistics cube

Return type iris.cube.Cube

esmvalcore.preprocessor.**multi_model_statistics**(*products*, *span*, *statistics*, *output_products=None*, *keep_input_datasets=True*)

Compute multi-model statistics.

This function computes multi-model statistics on a list of `products`, which can be instances of `Cube` or `PreprocessorFile`. The latter is used internally by ESMValCore to store workflow and provenance information, and this option should typically be ignored.

Apart from the time coordinate, cubes must have consistent shapes. There are two options to combine time coordinates of different lengths, see the `span` argument.

Uses the statistical operators in `iris.analysis`, including `mean`, `median`, `min`, `max`, and `std`. Percentiles are also supported and can be specified like `pXX.YY` (for percentile `XX.YY`; decimal part optional).

Notes

Some of the operators in `iris.analysis` require additional arguments. Except for percentiles, these operators are currently not supported.

Parameters

- **products** (*list*) – Cubes (or products) over which the statistics will be computed.
- **statistics** (*list*) – Statistical metrics to be computed, e.g. [mean, max]. Choose from the operators listed in the iris.analysis package. Percentiles can be specified like pXX.YY.
- **span** (*str*) – Overlap or full; if overlap, statitstics are computed on common time- span; if full, statistics are computed on full time spans, ignoring missing data.
- **output_products** (*dict*) – For internal use only. A dict with statistics names as keys and preprocessorfiles as values. If products are passed as input, the statistics cubes will be assigned to these output products.
- **keep_input_datasets** (*bool*) – If True, the output will include the input datasets. If False, only the computed statistics will be returned.

Returns A dictionary of statistics cubes with statistics' names as keys. (If input type is products, then it will return a set of output_products.)

Return type dict

Raises **ValueError** – If span is neither overlap nor full, or if input type is neither cubes nor products.

esmvalcore.preprocessor.**regrid**(*cube*, *target_grid*, *scheme*, *lat_offset=True*, *lon_offset=True*)

Perform horizontal regridding.

Note that the target grid can be a cube (`Cube`), path to a cube (`str`), a grid spec (`str`) in the form of *MxN*, or a `dict` specifying the target grid.

For the latter, the `target_grid` should be a `dict` with the following keys:

- `start_longitude`: longitude at the center of the first grid cell.
- `end_longitude`: longitude at the center of the last grid cell.
- **`step_longitude`: constant longitude distance between grid cell** centers.
- `start_latitude`: latitude at the center of the first grid cell.
- `end_latitude`: longitude at the center of the last grid cell.
- `step_latitude`: constant latitude distance between grid cell centers.

Parameters

- **cube** (`Cube`) – The source cube to be regridded.
- **target_grid** (*Cube or str or dict*) – The (location of a) cube that specifies the target or reference grid for the regridding operation.

  Alternatively, a string cell specification may be provided, of the form `MxN`, which specifies the extent of the cell, longitude by latitude (degrees) for a global, regular target grid.

  Alternatively, a dictionary with a regional target grid may be specified (see above).
- **scheme** (*str*) – The regridding scheme to perform, choose from `linear`, `linear_extrapolate`, `nearest`, `area_weighted`, `unstructured_nearest`.
- **lat_offset** (*bool*) – Offset the grid centers of the latitude coordinate w.r.t. the pole by half a grid step. This argument is ignored if `target_grid` is a cube or file.

- **lon_offset** (*bool*) – Offset the grid centers of the longitude coordinate w.r.t. Greenwich meridian by half a grid step. This argument is ignored if `target_grid` is a cube or file.

Returns Regridded cube.

Return type Cube

See also:

extract_levels Perform vertical regridding.

esmvalcore.preprocessor.**regrid_time**(*cube*, *frequency*)

Align time axis for cubes so they can be subtracted.

Operations on time units, time points and auxiliary coordinates so that any cube from cubes can be subtracted from any other cube from cubes. Currently this function supports yearly (frequency=yr), monthly (frequency=mon), daily (frequency=day), 6-hourly (frequency=6hr), 3-hourly (frequency=3hr) and hourly (frequency=1hr) data time frequencies.

Parameters

- **cube** (*iris.cube.Cube*) – input cube.
- **frequency** (*str*) – data frequency: mon, day, 1hr, 3hr or 6hr

Returns cube with converted time axis and units.

Return type iris.cube.Cube

esmvalcore.preprocessor.**remove_fx_variables**(*cube*)

Remove fx variables present as cell measures or ancillary variables in the cube containing the data.

Parameters **cube** (*iris.cube.Cube*) – Iris cube with data and cell measures or ancillary variables.

Returns Cube without cell measures or ancillary variables.

Return type iris.cube.Cube

esmvalcore.preprocessor.**resample_hours**(*cube*, *interval*, *offset=0*)

Convert x-hourly data to y-hourly by eliminating extra timesteps.

Convert x-hourly data to y-hourly (y > x) by eliminating the extra timesteps. This is intended to be used only with instantaneous values.

For example:

- resample_hours(cube, interval=6): Six-hourly intervals at 0:00, 6:00, 12:00, 18:00.
- resample_hours(cube, interval=6, offset=3): Six-hourly intervals at 3:00, 9:00, 15:00, 21:00.
- resample_hours(cube, interval=12, offset=6): Twelve-hourly intervals at 6:00, 18:00.

Parameters

- **cube** (*iris.cube.Cube*) – Input cube.
- **interval** (*int*) – The period (hours) of the desired data.
- **offset** (*int, optional*) – The firs hour (hours) of the desired data.

Returns Cube with the new frequency.

Return type iris.cube.Cube

Raises **ValueError:** – The specified frequency is not a divisor of 24.

esmvalcore.preprocessor.**resample_time**(*cube*, *month=None*, *day=None*, *hour=None*)

Change frequency of data by resampling it.

Converts data from one frequency to another by extracting the timesteps that match the provided month, day and/or hour. This is meant to be used with instantaneous values when computing statistics is not desired.

For example:

- resample_time(cube, hour=6): Daily values taken at 6:00.
- resample_time(cube, day=15, hour=6): Monthly values taken at 15th 6:00.
- resample_time(cube, month=6): Yearly values, taking in June
- resample_time(cube, month=6, day=1): Yearly values, taking 1st June

The condition must yield only one value per interval: the last two samples above will produce yearly data, but the first one is meant to be used to sample from monthly output and the second one will work better with daily.

Parameters

- **cube** (*iris.cube.Cube*) – Input cube.
- **month** (*int, optional*) – Month to extract
- **day** (*int, optional*) – Day to extract
- **hour** (*int, optional*) – Hour to extract

Returns Cube with the new frequency.

Return type iris.cube.Cube

esmvalcore.preprocessor.**save**(*cubes*, *filename*, *optimize_access=''*, *compress=False*, *alias=''*, ***kwargs*)

Save iris cubes to file.

Parameters

- **cubes** (*iterable of iris.cube.Cube*) – Data cubes to be saved
- **filename** (*str*) – Name of target file
- **optimize_access** (*str*) – Set internal NetCDF chunking to favour a reading scheme

  Values can be map or timeseries, which improve performance when reading the file one map or time series at a time. Users can also provide a coordinate or a list of coordinates. In that case the better performance will be avhieved by loading all the values in that coordinate at a time
- **compress** (*bool, optional*) – Use NetCDF internal compression.

Returns filename

Return type str

Raises **ValueError** – cubes is empty.

esmvalcore.preprocessor.**seasonal_statistics**(*cube*, *operator='mean'*, *seasons=('DJF', 'MAM', 'JJA', 'SON')*)

Compute seasonal statistics.

Chunks time seasons and computes statistics over them.

Parameters

- **cube** (*iris.cube.Cube*) – input cube.

- **operator** (*str, optional*) – Select operator to apply. Available operators: 'mean', 'median', 'std_dev', 'sum', 'min', 'max', 'rms'
- **seasons** (*list or tuple of str, optional*) – Seasons to build. Available: ('DJF', 'MAM', 'JJA', SON') (default) and all sequentially correct combinations holding every month of a year: e.g. ('JJAS','ONDJFMAM'), or less in case of prior season extraction.

Returns Seasonal statistic cube

Return type iris.cube.Cube

esmvalcore.preprocessor.**timeseries_filter**(*cube*, *window*, *span*, *filter_type='lowpass'*, *filter_stats='sum'*)

Apply a timeseries filter.

Method borrowed from iris example

Apply each filter using the rolling_window method used with the weights keyword argument. A weighted sum is required because the magnitude of the weights are just as important as their relative sizes.

See also the iris rolling window `iris.cube.Cube.rolling_window`.

Parameters

- **cube** (*iris.cube.Cube*) – input cube.
- **window** (*int*) – The length of the filter window (in units of cube time coordinate).
- **span** (*int*) – Number of months/days (depending on data frequency) on which weights should be computed e.g. 2-yearly: span = 24 (2 x 12 months). Span should have same units as cube time coordinate.
- **filter_type** (*str, optional*) – Type of filter to be applied; default 'lowpass'. Available types: 'lowpass'.
- **filter_stats** (*str, optional*) – Type of statistic to aggregate on the rolling window; default 'sum'. Available operators: 'mean', 'median', 'std_dev', 'sum', 'min', 'max', 'rms'

Returns cube time-filtered using 'rolling_window'.

Return type iris.cube.Cube

Raises

- **iris.exceptions.CoordinateNotFoundError:** – Cube does not have time coordinate.
- **NotImplementedError:** – If filter_type is not implemented.

esmvalcore.preprocessor.**volume_statistics**(*cube*, *operator*)

Apply a statistical operation over a volume.

The volume average is weighted according to the cell volume. Cell volume is calculated from iris's cartography tool multiplied by the cell thickness.

Parameters

- **cube** (*iris.cube.Cube*) – Input cube.
- **operator** (*str*) – The operation to apply to the cube, options are: 'mean'.

Returns collapsed cube.

Return type iris.cube.Cube

Raises `ValueError` – if input cube shape differs from grid volume cube shape.

esmvalcore.preprocessor.**weighting_landsea_fraction**(*cube*, *area_type*)
Weight fields using land or sea fraction.

This preprocessor function weights a field with its corresponding land or sea area fraction (value between 0 and 1). The application of this is important for most carbon cycle variables (and other land-surface outputs), which are e.g. reported in units of *kgC m-2*. This actually refers to 'per square meter of land/sea' and NOT 'per square meter of gridbox'. So in order to integrate these globally or regionally one has to both area-weight the quantity but also weight by the land/sea fraction.

Parameters

- **cube** (*iris.cube.Cube*) – Data cube to be weighted.
- **area_type** (*str*) – Use land (`'land'`) or sea (`'sea'`) fraction for weighting.

Returns Land/sea fraction weighted cube.

Return type iris.cube.Cube

Raises

- **TypeError** – `area_type` is not `'land'` or `'sea'`.
- **ValueError** – Land/sea fraction variables `sftlf` or `sftof` not found.

esmvalcore.preprocessor.**zonal_statistics**(*cube*, *operator*)
Compute zonal statistics.

Parameters

- **cube** (*iris.cube.Cube*) – input cube.
- **operator** (*str, optional*) – Select operator to apply. Available operators: 'mean', 'median', 'std_dev', 'sum', 'min', 'max', 'rms'.

Returns Zonal statistics cube.

Return type iris.cube.Cube

Raises **ValueError** – Error raised if computation on irregular grids is attempted. Zonal statistics not yet implemented for irregular grids.

CHAPTER

TWENTYNINE

EXPERIMENTAL API

This page describes the new ESMValCore API. The API module is available in the submodule `esmvalcore.experimental`. The API is under development, so use at your own risk!

29.1 Configuration

This section describes the *config* submodule of the API (`esmvalcore.experimental`).

29.1.1 Config

Configuration of ESMValCore/Tool is done via the *Config* object. The global configuration can be imported from the `esmvalcore.experimental` module as `CFG`:

```
>>> from esmvalcore.experimental import CFG
>>> CFG
Config({'auxiliary_data_dir': PosixPath('/home/user/auxiliary_data'),
        'compress_netcdf': False,
        'config_developer_file': None,
        'config_file': PosixPath('/home/user/.esmvaltool/config-user.yml'),
        'drs': {'CMIP5': 'default', 'CMIP6': 'default'},
        'exit_on_warning': False,
        'log_level': 'info',
        'max_parallel_tasks': None,
        'output_dir': PosixPath('/home/user/esmvaltool_output'),
        'output_file_type': 'png',
        'profile_diagnostic': False,
        'remove_preproc_dir': True,
        'rootpath': {'CMIP5': '~/default_inputpath',
                     'CMIP6': '~/default_inputpath',
                     'default': '~/default_inputpath'},
        'save_intermediary_cubes': False,
        'write_netcdf': True,
        'write_plots': True})
```

The parameters for the user configuration file are listed *here*.

`CFG` is essentially a python dictionary with a few extra functions, similar to `matplotlib.rcParams`. This means that values can be updated like this:

```
>>> CFG['output_dir'] = '~/esmvaltool_output'
>>> CFG['output_dir']
PosixPath('/home/user/esmvaltool_output')
```

Notice that `CFG` automatically converts the path to an instance of `pathlib.Path` and expands the home directory. All values entered into the config are validated to prevent mistakes, for example, it will warn you if you make a typo in the key:

```
>>> CFG['output_directory'] = '~/esmvaltool_output'
InvalidConfigParameter: `output_directory` is not a valid config parameter.
```

Or, if the value entered cannot be converted to the expected type:

```
>>> CFG['max_parallel_tasks'] = ''
InvalidConfigParameter: Key `max_parallel_tasks`: Could not convert '' to int
```

Config is also flexible, so it tries to correct the type of your input if possible:

```
>>> CFG['max_parallel_tasks'] = '8'  # str
>>> type(CFG['max_parallel_tasks'])
int
```

By default, the config is loaded from the default location (`/home/user/.esmvaltool/config-user.yml`). If it does not exist, it falls back to the default values. to load a different file:

```
>>> CFG.load_from_file('~/my-config.yml')
```

Or to reload the current config:

```
>>> CFG.reload()
```

29.1.2 Session

Recipes and diagnostics will be run in their own directories. This behaviour can be controlled via the *Session* object. A *Session* can be initiated from the global *Config*.

```
>>> session = CFG.start_session(name='my_session')
```

A *Session* is very similar to the config. It is also a dictionary, and copies all the keys from the *Config*. At this moment, `session` is essentially a copy of `CFG`:

```
>>> print(session == CFG)
True
>>> session['output_dir'] = '~/my_output_dir'
>>> print(session == CFG)  # False
False
```

A *Session* also knows about the directories where the data will stored. The session name is used to prefix the directories.

```
>>> session.session_dir
/home/user/my_output_dir/my_session_20201203_155821
>>> session.run_dir
```

(continues on next page)

(continued from previous page)

```
/home/user/my_output_dir/my_session_20201203_155821/run
>>> session.work_dir
/home/user/my_output_dir/my_session_20201203_155821/work
>>> session.preproc_dir
/home/user/my_output_dir/my_session_20201203_155821/preproc
>>> session.plot_dir
/home/user/my_output_dir/my_session_20201203_155821/plots
```

Unlike the global configuration, of which only one can exist, multiple sessions can be initiated from *Config*.

29.1.3 API reference

ESMValTool config module.

esmvalcore.experimental.config.**CFG**
ESMValCore configuration. By default this will loaded from the file ~/.esmvaltool/config-user.yml.

Classes:

Config(*args, **kwargs)	ESMValTool configuration object.
Session(config, name)	Container class for session configuration and directory information.

class esmvalcore.experimental.config.**Config**(**args*, ***kwargs*)
ESMValTool configuration object.

Do not instantiate this class directly, but use `esmvalcore.experimental.CFG` instead.

Methods:

load_from_file(filename)	Load user configuration from the given file.
reload()	Reload the config file.
start_session(name)	Start a new session from this configuration object.

load_from_file(*filename: Union[os.PathLike, str]*)
Load user configuration from the given file.

reload()
Reload the config file.

start_session(*name: str*)
Start a new session from this configuration object.

Parameters **name** (*str*) – Name of the session.

Returns

Return type *Session*

class esmvalcore.experimental.config.**Session**(*config: dict*, *name: str = 'session'*)
Container class for session configuration and directory information.

Do not instantiate this class directly, but use `CFG.start_session` instead.

Parameters

- **config** (*dict*) – Dictionary with configuration settings.

- **name** (*str*) – Name of the session to initialize, for example, the name of the recipe (default='session').

Attributes:

config_dir	Return user config directory.
main_log	Return main log file.
main_log_debug	Return main log debug file.
plot_dir	Return plot directory.
preproc_dir	Return preproc directory.
relative_main_log	
relative_main_log_debug	
relative_plot_dir	
relative_preproc_dir	
relative_run_dir	
relative_work_dir	
run_dir	Return run directory.
session_dir	Return session directory.
work_dir	Return work directory.

Methods:

from_config_user(config_user)	Convert *config-user* dict to API-compatible *Session* object.
set_session_name([name])	Set the name for the session.
to_config_user()	Turn the *Session* object into a recipe-compatible dict.

property config_dir
Return user config directory.

classmethod from_config_user(*config_user: dict*) → *esmvalcore.experimental.config._config_object.Session*
Convert *config-user* dict to API-compatible *Session* object.

For example, *_recipe.Recipe._cfg*.

property main_log
Return main log file.

property main_log_debug
Return main log debug file.

property plot_dir
Return plot directory.

property preproc_dir
Return preproc directory.

relative_main_log = PosixPath('run/main_log.txt')

relative_main_log_debug = PosixPath('run/main_log_debug.txt')

relative_plot_dir = PosixPath('plots')

relative_preproc_dir = PosixPath('preproc')

relative_run_dir = PosixPath('run')

relative_work_dir = PosixPath('work')

property run_dir
Return run directory.

property session_dir
Return session directory.

session_name: Optional[str]

set_session_name(*name: str = 'session'*)
Set the name for the session.

The *name* is used to name the session directory, e.g. *session_20201208_132800/*. The date is suffixed automatically.

to_config_user() → dict
Turn the *Session* object into a recipe-compatible dict.

This dict is compatible with the *config-user* argument in `esmvalcore._recipe.Recipe`.

property work_dir
Return work directory.

29.2 Recipes

This section describes the *recipe* submodule of the API (`esmvalcore.experimental`).

29.2.1 Recipe metadata

Recipe is a class that holds metadata from a recipe.

```
>>> Recipe('path/to/recipe_python.yml')
recipe = Recipe('Recipe Python')
```

Printing the recipe will give a nice overview of the recipe:

```
>>> print(recipe)
## Recipe python

Example recipe that plots a map and timeseries of temperature.

### Authors
 - Bouwe Andela (NLeSC, Netherlands; https://orcid.org/0000-0001-9005-8940)
 - Mattia Righi (DLR, Germany; https://orcid.org/0000-0003-3827-5950)

### Maintainers
 - Manuel Schlund (DLR, Germany; https://orcid.org/0000-0001-5251-0158)

### Projects
```

(continues on next page)

(continued from previous page)

```
 - DLR project ESMVal
 - Copernicus Climate Change Service 34a Lot 2 (MAGIC) project

### References
 - Please acknowledge the project(s).
```

29.2.2 Running a recipe

To run the recipe, call the *run()* method.

```
>>> output = recipe.run()
<log messages>
```

By default, a new *Session* is automatically created, so that data are never overwritten. Data are stored in the `esmvaltool_output` directory specified in the config. Sessions can also be explicitly specified.

```
>>> from esmvalcore.experimental import CFG
>>> session = CFG.start_session('my_session')
>>> output = recipe.run(session)
<log messages>
```

run() returns an dictionary of objects that can be used to inspect the output of the recipe. The output is an instance of *ImageFile* or *DataFile* depending on its type.

For working with recipe output, see: *Recipe output*.

29.2.3 Running a single diagnostic or preprocessor task

The python example recipe contains 5 tasks:

Preprocessors:

- `timeseries/tas_amsterdam`
- `timeseries/script1`
- `map/tas`

Diagnostics:

- `timeseries/tas_global`
- `map/script1`

To run a single diagnostic or preprocessor, the name of the task can be passed as an argument to *run()*. If a diagnostic is passed, all ancestors will automatically be run too.

```
>>> output = recipe.run('map/script1')
>>> output
map/script1:
  DataFile('CMIP5_CanESM2_Amon_historical_r1i1p1_tas_2000-2000.nc')
  DataFile('CMIP6_BCC-ESM1_Amon_historical_r1i1p1f1_tas_2000-2000.nc')
  ImageFile('CMIP5_CanESM2_Amon_historical_r1i1p1_tas_2000-2000.png')
  ImageFile('CMIP6_BCC-ESM1_Amon_historical_r1i1p1f1_tas_2000-2000.png')
```

It is also possible to run a single preprocessor task:

```
>>> output = recipe.run('map/tas')
>>> output
map/tas:
  DataFile('CMIP5_CanESM2_Amon_historical_r1i1p1_tas_2000-2000.nc')
  DataFile('CMIP6_BCC-ESM1_Amon_historical_r1i1p1f1_tas_2000-2000.nc')
```

29.2.4 API reference

Recipe metadata.

Classes:

Recipe(path)	API wrapper for the esmvalcore Recipe object.

class esmvalcore.experimental.recipe.**Recipe**(*path: os.PathLike*)

Bases: object

API wrapper for the esmvalcore Recipe object.

This class can be used to inspect and run the recipe.

Parameters **path** (*pathlike*) – Path to the recipe.

Attributes:

data	Return dictionary representation of the recipe.
name	Return the name of the recipe.

Methods:

get_output()	Get output from recipe.
render([template])	Render output as html.
run([task, session])	Run the recipe.

property data: dict

Return dictionary representation of the recipe.

get_output() → *esmvalcore.experimental.recipe_output.RecipeOutput*

Get output from recipe.

Returns **output** – Returns output of the recipe as instances of `OutputFile` grouped by diagnostic task.

Return type dict

property name

Return the name of the recipe.

render(*template=None*)

Render output as html.

template [Template] An instance of `jinja2.Template` can be passed to customize the output.

run(*task: Optional[str] = None*, *session: Optional[esmvalcore.experimental.config._config_object.Session] = None*)

Run the recipe.

This function loads the recipe into the ESMValCore recipe format and runs it.

Parameters

- **task** (*str*) – Specify the name of the diagnostic or preprocessor to run a single task.
- **session** (Session, optional) – Defines the config parameters and location where the recipe output will be stored. If None, a new session will be started automatically.

Returns output – Returns output of the recipe as instances of OutputItem grouped by diagnostic task.

Return type dict

29.3 Recipe output

This section describes the *recipe_output* submodule of the API (esmvalcore.experimental).

After running a recipe, output is returned by the *run()* method. Alternatively, it can be retrieved using the *get_output()* method.

```
>>> recipe_output = recipe.get_output()
```

recipe_output is a mapping of the individual tasks and their output filenames (data and image files) with a set of attributes describing the data.

```
>>> recipe_output
timeseries/script1:
  DataFile('tas_amsterdam_CMIP5_CanESM2_Amon_historical_r1i1p1_tas_1850-2000.nc')
  DataFile('tas_amsterdam_CMIP6_BCC-ESM1_Amon_historical_r1i1p1f1_tas_1850-2000.nc')
  DataFile('tas_amsterdam_MultiModelMean_Amon_tas_1850-2000.nc')
  DataFile('tas_global_CMIP5_CanESM2_Amon_historical_r1i1p1_tas_1850-2000.nc')
  DataFile('tas_global_CMIP6_BCC-ESM1_Amon_historical_r1i1p1f1_tas_1850-2000.nc')
  ImageFile('tas_amsterdam_CMIP5_CanESM2_Amon_historical_r1i1p1_tas_1850-2000.png')
  ImageFile('tas_amsterdam_CMIP6_BCC-ESM1_Amon_historical_r1i1p1f1_tas_1850-2000.png')
  ImageFile('tas_amsterdam_MultiModelMean_Amon_tas_1850-2000.png')
  ImageFile('tas_global_CMIP5_CanESM2_Amon_historical_r1i1p1_tas_1850-2000.png')
  ImageFile('tas_global_CMIP6_BCC-ESM1_Amon_historical_r1i1p1f1_tas_1850-2000.png')

map/script1:
  DataFile('CMIP5_CanESM2_Amon_historical_r1i1p1_tas_2000-2000.nc')
  DataFile('CMIP6_BCC-ESM1_Amon_historical_r1i1p1f1_tas_2000-2000.nc')
  ImageFile('CMIP5_CanESM2_Amon_historical_r1i1p1_tas_2000-2000.png')
  ImageFile('CMIP6_BCC-ESM1_Amon_historical_r1i1p1f1_tas_2000-2000.png')
```

Output is grouped by the task that produced them. They can be accessed like a dictionary.

```
>>> task_output = recipe_output['map/script1']
>>> task_output
map/script1:
  DataFile('CMIP5_CanESM2_Amon_historical_r1i1p1_tas_2000-2000.nc')
  DataFile('CMIP6_BCC-ESM1_Amon_historical_r1i1p1f1_tas_2000-2000.nc')
  ImageFile('CMIP5_CanESM2_Amon_historical_r1i1p1_tas_2000-2000.png')
  ImageFile('CMIP6_BCC-ESM1_Amon_historical_r1i1p1f1_tas_2000-2000.png')
```

The task output has a list of files associated with them, usually image (`.png`) or data files (`.nc`). To get a list of all files, use `files()`.

```
>>> print(task_output.files)
(DataFile('CMIP5_CanESM2_Amon_historical_r1i1p1_tas_2000-2000.nc'),
..., ImageFile('CMIP6_BCC-ESM1_Amon_historical_r1i1p1f1_tas_2000-2000.png'))
```

It is also possible to select the image (*image_files()*) files or data files (*data_files()*) only.

```
>>> for image_file in task_output.image_files:
>>>     print(image_file)
ImageFile('CMIP5_CanESM2_Amon_historical_r1i1p1_tas_2000-2000.png')
ImageFile('CMIP6_BCC-ESM1_Amon_historical_r1i1p1f1_tas_2000-2000.png')

>>> for data_file in task_output.data_files:
>>>     print(data_file)
DataFile('CMIP5_CanESM2_Amon_historical_r1i1p1_tas_2000-2000.nc')
DataFile('CMIP6_BCC-ESM1_Amon_historical_r1i1p1f1_tas_2000-2000.nc')
```

29.3.1 Working with output files

Output comes in two kinds, *DataFile* corresponds to data files in `.nc` format and *ImageFile* corresponds to plots in `.png` format (see below). Both object are derived from the same base class (*OutputFile*) and therefore share most of the functionality.

For example, author information can be accessed as instances of *Contributor* via

```
>>> output_file = task_output[0]
>>> output_file.authors
(Contributor('Andela, Bouwe', institute='NLeSC, Netherlands', orcid='https://orcid.org/
↪0000-0001-9005-8940'),
 Contributor('Righi, Mattia', institute='DLR, Germany', orcid='https://orcid.org/0000-
↪0003-3827-5950'))
```

And associated references as instances of *Reference* via

```
>>> output_file.references
(Reference('acknow_project'),)
```

OutputFile also knows about associated files

```
>>> data_file.citation_file
Path('.../tas_global_CMIP6_BCC-ESM1_Amon_historical_r1i1p1f1_tas_1850-2000_citation.
↪bibtex')
>>> data_file.data_citation_file
Path('.../tas_global_CMIP6_BCC-ESM1_Amon_historical_r1i1p1f1_tas_1850-2000_data_citation_
↪info.txt')
>>> data_file.provenance_svg_file
Path('.../tas_global_CMIP6_BCC-ESM1_Amon_historical_r1i1p1f1_tas_1850-2000_provenance.svg
↪')
>>> data_file.provenance_xml_file
Path('.../tas_global_CMIP6_BCC-ESM1_Amon_historical_r1i1p1f1_tas_1850-2000_provenance.xml
↪')
```

29.3.2 Working with image files

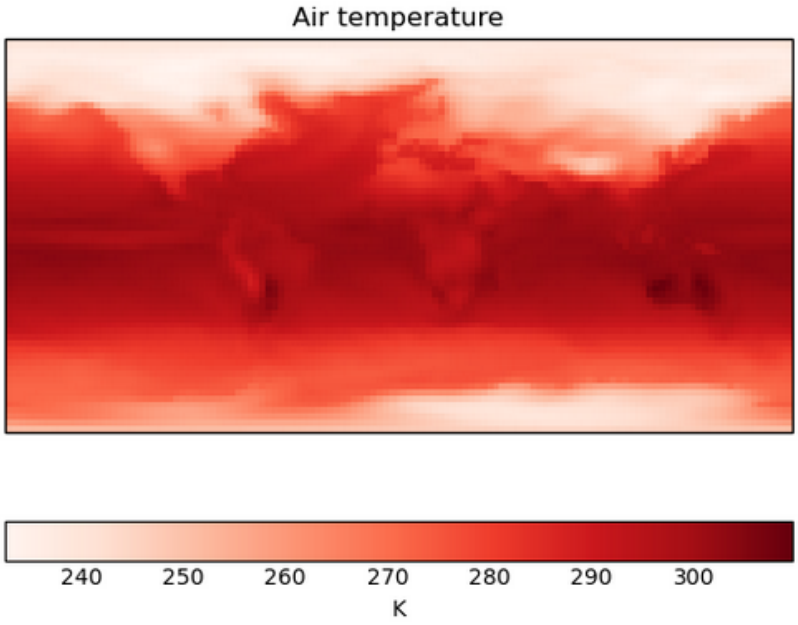
Image output uses IPython magic to plot themselves in a notebook environment.

```
>>> image_file = recipe_output['map/script1'].image_files[0]
>>> image_file
```

For example:

```
In [30]: image_file = recipe_output['map/script1'].image_files[0]
         image_file
```

Out[30]: Average Near-Surface Air Temperature between 2000 and 2000 according to BCC-ESM1.



Using `IPython.display`, it is possible to show all image files.

```
>>> from IPython.display import display
>>>
>>> task = recipe_output['map/script1']
>>> for image_file in task.image_files:
>>>     display(image_file)
```

29.3.3 Working with data files

Data files can be easily loaded using `xarray`:

```
>>> data_file = recipe_output['timeseries/script1'].data_files[0]
>>> data = data_file.load_xarray()
>>> type(data)
xarray.core.dataset.Dataset
```

Or `iris`:

```
>>> cube = data_file.load_iris()
>>> type(cube)
iris.cube.CubeList
```

29.3.4 API reference

API for handing recipe output.

Classes:

DataFile(path, attributes)	Container for data output.
ImageFile(path, attributes)	Container for image output.
OutputFile(path, attributes)	Base container for recipe output files.
RecipeOutput(task_output[, session, info])	Container for recipe output.
TaskOutput(name, files)	Container for task output.

class esmvalcore.experimental.recipe_output.**DataFile**(*path: str*, *attributes: Optional[dict] = None*)

Bases: *esmvalcore.experimental.recipe_output.OutputFile*

Container for data output.

Attributes:

authors	List of recipe authors.
caption	Return the caption of the file (fallback to path).
citation_file	Return path of citation file (bibtex format).
data_citation_file	Return path of data citation info (txt format).
kind	
provenance_svg_file	Return path of provenance file (svg format).
provenance_xml_file	Return path of provenance file (xml format).
references	List of project references.

Methods:

create(path[, attributes])	Construct new instances of OutputFile.
load_iris()	Load data using iris.
load_xarray()	Load data using xarray.

property authors: tuple

List of recipe authors.

property caption: str
Return the caption of the file (fallback to path).

property citation_file
Return path of citation file (bibtex format).

classmethod create(*path: str*, *attributes: Optional[dict] = None*) → *esmvalcore.experimental.recipe_output.OutputFile*
Construct new instances of OutputFile.

Chooses a derived class if suitable.

property data_citation_file
Return path of data citation info (txt format).

kind: Optional[str] = 'data'

load_iris()
Load data using iris.

load_xarray()
Load data using xarray.

property provenance_svg_file
Return path of provenance file (svg format).

property provenance_xml_file
Return path of provenance file (xml format).

property references: tuple
List of project references.

class esmvalcore.experimental.recipe_output.**ImageFile**(*path: str*, *attributes: Optional[dict] = None*)
Bases: *esmvalcore.experimental.recipe_output.OutputFile*

Container for image output.

Attributes:

authors	List of recipe authors.
caption	Return the caption of the file (fallback to path).
citation_file	Return path of citation file (bibtex format).
data_citation_file	Return path of data citation info (txt format).
kind	
provenance_svg_file	Return path of provenance file (svg format).
provenance_xml_file	Return path of provenance file (xml format).
references	List of project references.

Methods:

create(path[, attributes])	Construct new instances of OutputFile.
to_base64()	Encode image as base64 to embed in a Jupyter notebook.

property authors: tuple
List of recipe authors.

property caption: str

Return the caption of the file (fallback to path).

property citation_file
Return path of citation file (bibtex format).

classmethod create(*path: str*, *attributes: Optional[dict] = None*) → *esmvalcore.experimental.recipe_output.OutputFile*
Construct new instances of OutputFile.

Chooses a derived class if suitable.

property data_citation_file
Return path of data citation info (txt format).

kind: Optional[str] = 'image'

property provenance_svg_file
Return path of provenance file (svg format).

property provenance_xml_file
Return path of provenance file (xml format).

property references: tuple
List of project references.

to_base64() → str
Encode image as base64 to embed in a Jupyter notebook.

class esmvalcore.experimental.recipe_output.**OutputFile**(*path: str*, *attributes: Optional[dict] = None*)
Bases: object

Base container for recipe output files.

Use *OutputFile.create(path='<path>', attributes=attributes)* to initialize a suitable subclass.

Parameters

- **path** (*str*) – Name of output file
- **attributes** (*dict*) – Attributes corresponding to the recipe output

Attributes:

authors	List of recipe authors.
caption	Return the caption of the file (fallback to path).
citation_file	Return path of citation file (bibtex format).
data_citation_file	Return path of data citation info (txt format).
kind	
provenance_svg_file	Return path of provenance file (svg format).
provenance_xml_file	Return path of provenance file (xml format).
references	List of project references.

Methods:

create(path[, attributes])	Construct new instances of OutputFile.

property authors: tuple
List of recipe authors.

property caption: str

Return the caption of the file (fallback to path).

property citation_file
Return path of citation file (bibtex format).

classmethod create(*path: str*, *attributes: Optional[dict] = None*) → *esmvalcore.experimental.recipe_output.OutputFile*
Construct new instances of OutputFile.

Chooses a derived class if suitable.

property data_citation_file
Return path of data citation info (txt format).

kind: Optional[str] = None

property provenance_svg_file
Return path of provenance file (svg format).

property provenance_xml_file
Return path of provenance file (xml format).

property references: tuple
List of project references.

class esmvalcore.experimental.recipe_output.**RecipeOutput**(*task_output: dict*, *session=None*, *info=None*)

Bases: collections.abc.Mapping

Container for recipe output.

Parameters **task_output** (*dict*) – Dictionary with recipe output grouped by task name. Each task value is a mapping of the filenames with the product attributes.

Methods:

from_core_recipe_output(recipe_output)	Construct instance from *_recipe.Recipe* output.
get(k[,d])	
items()	
keys()	
read_main_log()	Read log file.
read_main_log_debug()	Read debug log file.
render([template])	Render output as html.
values()	
write_html()	Write output summary to html document.

classmethod from_core_recipe_output(*recipe_output: dict*)
Construct instance from *_recipe.Recipe* output.

The core recipe format is not directly compatible with the API. This constructor does the following:

1. Convert *config-user* dict to an instance of `Session`
2. Converts the raw recipe dict to `RecipeInfo`

Parameters **recipe_output** (*dict*) – Output from *_recipe.Recipe.get_product_output*

get(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

items() → a set-like object providing a view on D's items

keys() → a set-like object providing a view on D's keys

read_main_log() → str
Read log file.

read_main_log_debug() → str
Read debug log file.

render(*template=None*)
Render output as html.

template [Template] An instance of `jinja2.Template` can be passed to customize the output.

values() → an object providing a view on D's values

write_html()
Write output summary to html document.

A html file *index.html* gets written to the session directory.

class esmvalcore.experimental.recipe_output.**TaskOutput**(*name: str*, *files: dict*)
Bases: object

Container for task output.

Parameters

- **name** (*str*) – Name of the task
- **files** (*dict*) – Mapping of the filenames with the associated attributes.

Attributes:

data_files	Return a tuple of data objects.
image_files	Return a tuple of image objects.

Methods:

from_task(task)	Create an instance of *TaskOutput* from a Task.

property data_files: tuple
Return a tuple of data objects.

classmethod from_task(*task*) → *esmvalcore.experimental.recipe_output.TaskOutput*
Create an instance of *TaskOutput* from a Task.

Where task is an instance of *esmvalcore._task.BaseTask*.

property image_files: tuple
Return a tuple of image objects.

29.4 Recipe Metadata

This section describes the *recipe_metadata* submodule of the API (`esmvalcore.experimental`).

29.4.1 API reference

API for recipe metadata.

Classes:

Contributor(name, institute, orcid)	Contains contributor (author or maintainer) information.
Project(project)	Use this class to acknowledge a project associated with the recipe.
Reference(filename)	Parse reference information from bibtex entries.

Exceptions:

RenderError	Error during rendering of object.

class esmvalcore.experimental.recipe_metadata.**Contributor**(*name: str*, *institute: str*, *orcid: Optional[str] = None*)

Bases: `object`

Contains contributor (author or maintainer) information.

Parameters

- **name** (*str*) – Name of the author, i.e. `'John Doe'`
- **institute** (*str*) – Name of the institute
- **orcid** (*str, optional*) – ORCID url

Methods:

from_dict(attributes)	Return an instance of Contributor from a dictionary.
from_tag(tag)	Return an instance of Contributor from a tag (`TAGS`).

classmethod from_dict(*attributes*)

Return an instance of Contributor from a dictionary.

Parameters **attributes** (*dict*) – Dictionary containing name / institute [/ orcid].

classmethod from_tag(*tag: str*) → *esmvalcore.experimental.recipe_metadata.Contributor*

Return an instance of Contributor from a tag (`TAGS`).

Parameters **tag** (*str*) – The contributor tags are defined in the authors section in `config-references.yml`.

class esmvalcore.experimental.recipe_metadata.**Project**(*project: str*)

Bases: `object`

Use this class to acknowledge a project associated with the recipe.

Parameters **project** (*str*) – The project title.

Methods:

from_tag(tag)	Return an instance of Project from a tag (TAGS).

classmethod from_tag(*tag: str*) → *esmvalcore.experimental.recipe_metadata.Project*
Return an instance of Project from a tag (TAGS).

Parameters **tag** (*str*) – The project tags are defined in config-references.yml.

class esmvalcore.experimental.recipe_metadata.**Reference**(*filename: str*)
Bases: object

Parse reference information from bibtex entries.

Parameters **filename** (*str*) – Name of the bibtex file.

Raises **NotImplementedError** – If the bibtex file contains more than 1 entry.

Methods:

from_tag(tag)	Return an instance of Reference from a bibtex tag.
render([renderer])	Render the reference.

classmethod from_tag(*tag: str*) → *esmvalcore.experimental.recipe_metadata.Reference*
Return an instance of Reference from a bibtex tag.

Parameters **tag** (*str*) – The bibtex tags resolved as esmvaltool/references/{tag}.bibtex or the corresponding directory as defined by the diagnostics path.

render(*renderer: str = 'html'*) → str
Render the reference.

Parameters **renderer** (*str*) – Choose the renderer for the string representation. Must be one of: 'plaintext', 'markdown', 'html', 'latex'

Returns Rendered reference

Return type str

exception esmvalcore.experimental.recipe_metadata.**RenderError**
Bases: BaseException

Error during rendering of object.

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

29.5 Utils

This section describes the *utils* submodule of the API (esmvalcore.experimental).

29.5.1 Finding recipes

One of the first thing we may want to do, is to simply get one of the recipes available in `ESMValTool`

If you already know which recipe you want to load, call *get_recipe()*.

```
from esmvalcore.experimental import get_recipe
>>> get_recipe('examples/recipe_python')
Recipe('Recipe python')
```

Call the *get_all_recipes()* function to get a list of all available recipes.

```
>>> from esmvalcore.experimental import get_all_recipes
>>> recipes = get_all_recipes()
>>> recipes
[Recipe('Recipe perfmetrics cmip5 4cds'),
 Recipe('Recipe martin18grl'),
 ...
 Recipe('Recipe wflow'),
 Recipe('Recipe pcrglobwb')]
```

To search for a specific recipe, you can use the *find()* method. This takes a search query that looks through the recipe metadata and returns any matches. The query can be a regex pattern, so you can make it as complex as you like.

```
>>> results = recipes.find('climwip')
[Recipe('Recipe climwip')]
```

The recipes are loaded in a *Recipe* object, which knows about the documentation, authors, project, and related references of the recipe. It resolves all the tags, so that it knows which institute an author belongs to and which references are associated with the recipe.

This means you can search for something like this:

```
>>> recipes.find('Geophysical Research Letters')
[Recipe('Recipe martin18grl'),
 Recipe('Recipe climwip'),
 Recipe('Recipe ecs constraints'),
 Recipe('Recipe ecs scatter'),
 Recipe('Recipe ecs'),
 Recipe('Recipe seaice')]
```

29.5.2 API reference

ESMValCore utilities.

Classes:

RecipeList([iterable])	Container for recipes.

Functions:

get_all_recipes([subdir])	Return a list of all available recipes.
get_recipe(name)	Get a recipe by its name.

class esmvalcore.experimental.utils.**RecipeList**(*iterable=()*, */*)

Container for recipes.

Methods:

find(query)	Search for recipes matching the search query or pattern.

find(*query: Pattern[str]*)

Search for recipes matching the search query or pattern.

Searches in the description, authors and project information fields. All matches are returned.

Parameters **query** (*str, Pattern*) – String to search for, e.g. `find_recipes('righi')` will return all matching that author. Can be a *regex* pattern.

Returns List of recipes matching the search query.

Return type *RecipeList*

esmvalcore.experimental.utils.**get_all_recipes**(*subdir: Optional[str] = None*) → list

Return a list of all available recipes.

Parameters **subdir** (*str*) – Sub-directory of the `DIAGNOSTICS.path` to look for recipes, e.g. `get_all_recipes(subdir='examples')`.

Returns List of available recipes

Return type *RecipeList*

esmvalcore.experimental.utils.**get_recipe**(*name: Union[os.PathLike, str]*) → *esmvalcore.experimental.recipe.Recipe*

Get a recipe by its name.

The function looks first in the local directory, and second in the repository defined by the diagnostic path. The recipe name can be specified with or without extension. The first match will be returned.

Parameters **name** (*str, pathlike*) – Name of the recipe file, i.e. `examples/recipe_python.yml`

Returns Instance of `Recipe` which can be used to inspect and run the recipe.

Return type *Recipe*

Raises `FileNotFoundError` – If the name cannot be resolved to a recipe file.

Part VII

Changelog

CHAPTER

THIRTY

V2.3.0

This release includes

30.1 Bug fixes

- Extend preprocessor multi_model_statistics to handle data with "altitude" coordinate (#1010) Axel Lauer
- Remove scripts included with CMOR tables (#1011) Bouwe Andela
- Avoid side effects in extract_season (#1019) Javier Vegas-Regidor
- Use nearest scheme to avoid interpolation errors with masked data in regression test (#1021) Stef Smeets
- Move _get_time_bounds from preprocessor._time to cmor.check to avoid circular import with cmor module (#1037) Valeriu Predoi
- Fix test that makes conda build fail (#1046) Valeriu Predoi
- Fix 'positive' attribute for rsns/rlns variables (#1051) Lukas Brunner
- Added preprocessor mask_multimodel (#767) Manuel Schlund
- Fix bug when fixing bounds after fixing longitude values (#1057) sloosvel
- Run conda build parallel AND sequential tests (#1065) Valeriu Predoi
- Add key to id_prop (#1071) Lukas Brunner
- Fix bounds after reversing coordinate values (#1061) sloosvel
- Fixed –skip-nonexistent option (#1093) Manuel Schlund
- Do not consider CMIP5 variable sit to be the same as sithick from CMIP6 (#1033) Bouwe Andela
- Improve finding date range in filenames (enforces separators) (#1145) Stéphane Sénési - work
- Review fx handling (#1147) sloosvel
- Fix lru cache decorator with explicit call to method (#1172) Valeriu Predoi
- Update _volume.py (#1174) Lee de Mora

30.2 Deprecations

30.3 Documentation

- Final changelog for 2.3.0 (#1163) Klaus Zimmermann
- Set version to 2.3.0 (#1162) Klaus Zimmermann
- Fix documentation build (#1006) Bouwe Andela
- Add labels required for linking from ESMValTool docs (#1038) Bouwe Andela
- Update contribution guidelines (#1047) Bouwe Andela
- Fix basestring references in documentation (#1106) Javier Vegas-Regidor
- Updated references master to main (#1132) Axel Lauer
- Add instructions how to use the central installation at DKRZ-Mistral (#1155) Rémi Kazeroni

30.4 Fixes for datasets

- Added fixes for various CMIP5 datasets, variable cl (3-dim cloud fraction) (#1017) Axel Lauer
- Added fixes for hybrid level coordinates of CESM2 models (#882) Manuel Schlund
- Extending LWP fix for CMIP6 models (#1049) Axel Lauer
- Add fixes for the net & up radiation variables from ERA5 (#1052) Lukas Brunner
- Add derived variable rsus (#1053) Lukas Brunner
- Supported *mip*-level fixes (#1095) Manuel Schlund
- Fix erroneous use of *grid_latitude* and *grid_longitude* and cleaned ocean grid fixes (#1092) Manuel Schlund
- Fix for pr of miroc5 (#1110) Rémi Kazeroni
- Ocean depth fix for cnrm_esm2_1, gfdl_esm4, ipsl_cm6a_lr datasets + mcm_ua_1_0 (#1098) Tomas Lovato
- Fix for uas variable of the MCM_UA_1_0 dataset (#1102) Rémi Kazeroni
- Fixes for sos and siconc of BCC models (#1090) Rémi Kazeroni
- Run fgco2 fix for all CESM2 models (#1108) Lisa Bock
- Fixes for the siconc variable of CMIP6 models (#1105) Rémi Kazeroni
- Fix wrong sign for land surface flux (#1113) Lisa Bock
- Fix for pr of EC_EARTH (#1116) Rémi Kazeroni

30.5 CMOR standard

- Format cmor related files (#976) Javier Vegas-Regidor
- Check presence of time bounds and guess them if needed (#849) sloosvel
- Add custom variable "tasaga" (#1118) Lisa Bock
- Find files for CMIP6 DCPP startdates (#771) sloosvel

30.6 Preprocessor

- Update tests for multimodel statistics preprocessor (#1023) Stef Smeets
- Raise in extract_season and extract_month if result is None (#1041) Javier Vegas-Regidor
- Allow selection of shapes in extract_shape (#764) Javier Vegas-Regidor
- Add option for regional regridding to regrid preprocessor (#1034) Stef Smeets
- Load fx variables as cube cell measures / ancillary variables (#999) sloosvel
- Check horizontal grid before regridding (#507) Benjamin Müller
- Clip irregular grids (#245) Bouwe Andela
- Use native iris functions in multi-model statistics (#1150) Peter Kalverla

30.7 Notebook API (experimental)

30.8 Automatic testing

- Report coverage for tests that run on any pull request (#994) Bouwe Andela
- Install ESMValTool sample data from PyPI (#998) Javier Vegas-Regidor
- Fix tests for multi-processing with spawn method (i.e. macOSX with Python>3.8) (#1003) Barbara Vreede
- Switch to running the Github Action test workflow every 3 hours in single thread mode to observe if Sementation Faults occur (#1022) Valeriu Predoi
- Revert to original Github Actions test workflow removing the 3-hourly test run with -n 1 (#1025) Valeriu Predoi
- Avoid stale cache for multimodel statistics regression tests (#1030) Bouwe Andela
- Add newer Python versions in OSX to Github Actions (#1035) Barbara Vreede
- Add tests for type annotations with mypy (#1042) Stef Smeets
- Run problematic cmor tests sequentially to avoid segmentation faults on CircleCI (#1064) Valeriu Predoi
- Test installation of esmvalcore from conda-forge (#1075) Valeriu Predoi
- Added additional test cases for integration tests of data_finder.py (#1087) Manuel Schlund
- Pin cf-units and fix tests (cf-units>=2.1.5) (#1140) Valeriu Predoi
- Fix failing CircleCI tests (#1167) Bouwe Andela
- Fix test failing due to fx files chosen differently on different OS's (#1169) Valeriu Predoi

- Compare datetimes instead of strings in _fixes/cmip5/test_access1_X.py (#1173) Valeriu Predoi
- Pin Python to 3.9 in environment.yml on CircleCI and skip mypy tests in conda build (#1176) Bouwe Andela

30.9 Installation

- Update yamale to version 3 (#1059) Klaus Zimmermann

30.10 Improvements

- Refactor diagnostics / tags management (#939) Stef Smeets
- Support multiple paths in input_dir (#1000) Javier Vegas-Regidor
- Generate HTML report with recipe output (#991) Stef Smeets
- Add timeout to requests.get in _citation.py (#1091) SarahAlidoost
- Add SYNDA drs for CMIP5 and CMIP6 (closes #582) (#583) Klaus Zimmermann
- Add basic support for variable mappings (#1124) Klaus Zimmermann
- Handle IPSL-CM6 (#1153) Stéphane Sénési - work

CHAPTER

THIRTYONE

V2.2.0

31.1 Highlights

ESMValCore is now using the recently released Iris 3. We acknowledge that this change may impact your work, as Iris 3 introduces several changes that are not backward-compatible, but we think that moving forward is the best decision for the tool in the long term.

This release is also the first one including support for downloading CMIP6 data using Synda and we have also started supporting Python 3.9. Give it a try!

This release includes

31.2 Bug fixes

- Fix path settings for DKRZ/Mistral (#852) Bouwe Andela
- Change logic for calling the diagnostic script to avoid problems with scripts where the executable bit is accidentally set (#877) Bouwe Andela
- Fix overwriting in generic level check (#886) sloosvel
- Add double quotes to script args in rerun screen message when using vprof profiling (#897) Valeriu Predoi
- Simplify time handling in multi-model statistics preprocessor (#685) Peter Kalverla
- Fix links to Iris documentation (#966) Javier Vegas-Regidor
- Bugfix: Fix units for MSWEP data (#986) Stef Smeets

31.3 Deprecations

- Deprecate defining write_plots and write_netcdf in config-user file (#808) Bouwe Andela

31.4 Documentation

- Fix numbering of steps in release instructions (#838) Bouwe Andela
- Add labels to changelogs of individual versions for easy reference (#899) Klaus Zimmermann
- Make CircleCI badge specific to main branch (#902) Bouwe Andela
- Fix docker build badge url (#906) Stef Smeets
- Update github PR template (#909) Stef Smeets
- Refer to ESMValTool GitHub discussions page in the error message (#900) Bouwe Andela
- Support automatically closing issues (#922) Bouwe Andela
- Fix checkboxes in PR template (#931) Stef Smeets
- Change in config-user defaults and documentation with new location for esmeval OBS data on JASMIN (#958) Valeriu Predoi
- Update Core Team info (#942) Axel Lauer
- Update iris documentation URL for sphinx (#964) Bouwe Andela
- Set version to 2.2.0 (#977) Javier Vegas-Regidor
- Add first draft of v2.2.0 changelog (#983) Javier Vegas-Regidor
- Add checkbox in PR template to assign labels (#985) Javier Vegas-Regidor
- Update install.rst (#848) bascrezee
- Change the order of the publication steps (#984) Javier Vegas-Regidor
- Add instructions how to use esmvaltool from HPC central installations (#841) Valeriu Predoi

31.5 Fixes for datasets

- Fixing unit for derived variable rsnstcsnorm to prevent overcorrection2 (#846) katjaweigel
- Cmip6 fix awi cm 1 1 mr (#822) mwjury
- Cmip6 fix ec earth3 veg (#836) mwjury
- Changed latitude longitude fix from Tas to AllVars. (#916) katjaweigel
- Fix for precipitation (pr) to use ERA5-Land cmorizer (#879) katjaweigel
- Cmip6 fix ec earth3 (#837) mwjury
- Cmip6_fix_fgoals_f3_l_Amon_time_bnds (#831) mwjury
- Fix for FGOALS-f3-L sftlf (#667) mwjury
- Improve ACCESS-CM2 and ACCESS-ESM1-5 fixes and add CIESM and CESM2-WACCM-FV2 fixes for cl, clw and cli (#635) Axel Lauer
- Add fixes for cl, cli, clw and tas for several CMIP6 models (#955) Manuel Schlund
- Dataset fixes for MSWEP (#969) Stef Smeets
- Dataset fixes for: ACCESS-ESM1-5, CanESM5, CanESM5 for carbon cycle (#947) Bettina Gier
- Fixes for KIOST-ESM (CMIP6) (#904) Rémi Kazeroni

- Fixes for AWI-ESM-1-1-LR (CMIP6, piControl) (#911) Rémi Kazeroni

31.6 CMOR standard

- CMOR check generic level coordinates in CMIP6 (#598) sloosvel
- Update CMIP6 tables to 6.9.33 (#919) Javier Vegas-Regidor
- Adding custom variables for tas uncertainty (#924) Lisa Bock
- Remove monotonicity coordinate check for unstructured grids (#965) Javier Vegas-Regidor

31.7 Preprocessor

- Added clip_start_end_year preprocessor (#796) Manuel Schlund
- Add support for downloading CMIP6 data with Synda (#699) Bouwe Andela
- Add multimodel tests using real data (#856) Stef Smeets
- Add plev/altitude conversion to extract_levels (#892) Axel Lauer
- Add possibility of custom season extraction. (#247) mwjury
- Adding the ability to derive xch4 (#783) Birgit Hassler
- Add preprocessor function to resample time and compute x-hourly statistics (#696) Javier Vegas-Regidor
- Fix duplication in preprocessors DEFAULT_ORDER introduced in #696 (#973) Javier Vegas-Regidor
- Use consistent precision in multi-model statistics calculation and update reference data for tests (#941) Peter Kalverla
- Refactor multi-model statistics code to facilitate ensemble stats and lazy evaluation (#949) Peter Kalverla
- Add option to exclude input cubes in output of multimodel statistics to solve an issue introduced by #949 (#978) Peter Kalverla

31.8 Automatic testing

- Pin cftime>=1.3.0 to have newer string formatting in and fix two tests (#878) Valeriu Predoi
- Switched miniconda conda setup hooks for Github Actions workflows (#873) Valeriu Predoi
- Add test for latest version resolver (#874) Stef Smeets
- Update codacy coverage reporter to fix coverage (#905) Niels Drost
- Avoid hardcoded year in tests and add improvement to plev test case (#921) Bouwe Andela
- Pin scipy to less than 1.6.0 until ESMValGroup/ESMValCore/issues/927 gets resolved (#928) Valeriu Predoi
- Github Actions: change time when conda install test runs (#930) Valeriu Predoi
- Remove redundant test line from test_utils.py (#935) Valeriu Predoi
- Removed netCDF4 package from integration tests of fixes (#938) Manuel Schlund
- Use new conda environment for installing ESMValCore in Docker containers (#951) Bouwe Andela

31.9 Notebook API (experimental)

- Implement importable config object in experimental API submodule (#868) Stef Smeets
- Add loading and running recipes to the notebook API (#907) Stef Smeets
- Add displaying and loading of recipe output to the notebook API (#957) Stef Smeets
- Add functionality to run single diagnostic task to notebook API (#962) Stef Smeets

31.10 Improvements

- Create CODEOWNERS file (#809) Javier Vegas-Regidor
- Remove code needed for Python <3.6 (#844) Bouwe Andela
- Add requests as a dependency (#850) Bouwe Andela
- Pin Python to less than 3.9 (#870) Valeriu Predoi
- Remove numba dependency (#880) Manuel Schlund
- Add Listing and finding recipes to the experimental notebook API (#901) Stef Smeets
- Skip variables that don't have dataset or additional_dataset keys (#860) Valeriu Predoi
- Refactor logging configuration (#933) Stef Smeets
- Xco2 derivation (#913) Bettina Gier
- Working environment for Python 3.9 (pin to !=3.9.0) (#885) Valeriu Predoi
- Print source file when using config get_config_user command (#960) Valeriu Predoi
- Switch to Iris 3 (#819) Stef Smeets
- Refactor tasks (#959) Stef Smeets
- Restore task summary in debug log after #959 (#981) Bouwe Andela
- Pin pre-commit hooks (#974) Stef Smeets
- Improve error messages when data is missing (#917) Javier Vegas-Regidor
- Set remove_preproc_dir to false in default config-user (#979) Valeriu Predoi
- Move fiona to be installed from conda forge (#987) Valeriu Predoi
- Re-added fiona in setup.py (#990) Valeriu Predoi

CHAPTER

THIRTYTWO

V2.1.0

This release includes

32.1 Bug fixes

- Set unit=1 if anomalies are standardized (#727) bascrezee
- Fix crash for FGOALS-g2 variables without longitude coordinate (#729) Bouwe Andela
- Improve variable alias management (#595) Javier Vegas-Regidor
- Fix area_statistics fx files loading (#798) Javier Vegas-Regidor
- Fix units after derivation (#754) Manuel Schlund

32.2 Documentation

- Update v2.0.0 release notes with final additions (#722) Bouwe Andela
- Update package description in setup.py (#725) Mattia Righi
- Add installation instructions for pip installation (#735) Bouwe Andela
- Improve config-user documentation (#740) Bouwe Andela
- Update the zenodo file with contributors (#807) Valeriu Predoi
- Improve command line run documentation (#721) Javier Vegas-Regidor
- Update the zenodo file with contributors (continued) (#810) Valeriu Predoi

32.3 Improvements

- Reduce size of docker image (#723) Javier Vegas-Regidor
- Add 'test' extra to installation, used by docker development tag (#733) Bouwe Andela
- Correct dockerhub link (#736) Bouwe Andela
- Create action-install-from-pypi.yml (#734) Valeriu Predoi
- Add pre-commit for linting/formatting (#766) Stef Smeets
- Run tests in parallel and when building conda package (#745) Bouwe Andela

- Readable exclude pattern for pre-commit (#770) Stef Smeets
- Github Actions Tests (#732) Valeriu Predoi
- Remove isort setup to fix formatting conflict with yapf (#778) Stef Smeets
- Fix yapf-isort import formatting conflict (Fixes #777) (#784) Stef Smeets
- Sorted output for *esmvaltool recipes list* (#790) Stef Smeets
- Replace vmprof with vprof (#780) Valeriu Predoi
- Update CMIP6 tables to 6.9.32 (#706) Javier Vegas-Regidor
- Default config-user path now set in config-user read function (#791) Javier Vegas-Regidor
- Add custom variable lweGrace (#692) bascrezee
- Create Github Actions workflow to build and deploy on Test PyPi and PyPi (#820) Valeriu Predoi
- Build and publish the esmvalcore package to conda via Github Actions workflow (#825) Valeriu Predoi

32.4 Fixes for datasets

- Fix cmip6 models (#629) npgillett
- Fix siconca variable in EC-Earth3 and EC-Earth3-Veg models in amip simulation (#702) Evgenia Galytska

32.5 Preprocessor

- Move cmor_check_data to early in preprocessing chain (#743) Bouwe Andela
- Add RMS iris analysis operator to statistics preprocessor functions (#747) Pep Cos
- Add surface chlorophyll concentration as a derived variable (#720) sloosvel
- Use dask to reduce memory consumption of extract_levels for masked data (#776) Valeriu Predoi

CHAPTER

THIRTYTHREE

V2.0.0

This release includes

33.1 Bug fixes

- Fixed derivation of co2s (#594) Manuel Schlund
- Padding while cropping needs to stay within sane bounds for shapefiles that span the whole Earth (#626) Valeriu Predoi
- Fix concatenation of a single cube (#655) Bouwe Andela
- Fix mask fx dict handling not to fail if empty list in values (#661) Valeriu Predoi
- Preserve metadata during anomalies computation when using iris cubes difference (#652) Valeriu Predoi
- Avoid crashing when there is directory 'esmvaltool' in the current working directory (#672) Valeriu Predoi
- Solve bug in ACCESS1 dataset fix for calendar. (#671) Peter Kalverla
- Fix the syntax for adding multiple ensemble members from the same dataset (#678) SarahAlidoost
- Fix bug that made preprocessor with fx files fail in rare cases (#670) Manuel Schlund
- Add support for string coordinates (#657) Javier Vegas-Regidor
- Fixed the shape extraction to account for wraparound shapefile coords (#319) Valeriu Predoi
- Fixed bug in time weights calculation (#695) Manuel Schlund
- Fix diagnostic filter (#713) Javier Vegas-Regidor

33.2 Documentation

- Add pandas as a requirement for building the documentation (#607) Bouwe Andela
- Document default order in which preprocessor functions are applied (#633) Bouwe Andela
- Add pointers about data loading and CF standards to documentation (#571) Valeriu Predoi
- Config file populated with site-specific data paths examples (#619) Valeriu Predoi
- Update Codacy badges (#643) Bouwe Andela
- Update copyright info on readthedocs (#668) Bouwe Andela
- Updated references to documentation (now docs.esmvaltool.org) (#675) Axel Lauer

- Add all European grants to Zenodo (#680) Bouwe Andela
- Update Sphinx to v3 or later (#683) Bouwe Andela
- Increase version to 2.0.0 and add release notes (#691) Bouwe Andela
- Update setup.py and README.md for use on PyPI (#693) Bouwe Andela
- Suggested Documentation changes (#690) Steve Smith

33.3 Improvements

- Reduce the size of conda package (#606) Bouwe Andela
- Add a few unit tests for DiagnosticTask (#613) Bouwe Andela
- Make ncl or R tests not fail if package not installed (#610) Valeriu Predoi
- Pin flake8<3.8.0 (#623) Valeriu Predoi
- Log warnings for likely errors in provenance record (#592) Bouwe Andela
- Unpin flake8 (#646) Bouwe Andela
- More flexible native6 default DRS (#645) Bouwe Andela
- Try to use the same python for running diagnostics as for esmvaltool (#656) Bouwe Andela
- Fix test for lower python version and add note on lxml (#659) Valeriu Predoi
- Added 1m deep average soil moisture variable (#664) bascrezee
- Update docker recipe (#603) Javier Vegas-Regidor
- Improve command line interface (#605) Javier Vegas-Regidor
- Remove utils directory (#697) Bouwe Andela
- Avoid pytest version that crashes (#707) Bouwe Andela
- Options arg in read_config_user_file now optional (#716) Javier Vegas-Regidor
- Produce a readable warning if ancestors are a string instead of a list. (#711) katjaweigel
- Pin Yamale to v2 (#718) Bouwe Andela
- Expanded cmor public API (#714) Manuel Schlund

33.4 Fixes for datasets

- Added various fixes for hybrid height coordinates (#562) Manuel Schlund
- Extended fix for cl-like variables of CESM2 models (#604) Manuel Schlund
- Added fix to convert "geopotential" to "geopotential height" for ERA5 (#640) Evgenia Galytska
- Do not fix longitude values if they are too far from valid range (#636) Javier Vegas-Regidor

33.5 Preprocessor

- Implemented concatenation of cubes with derived coordinates (#546) Manuel Schlund
- Fix derived variable ctotal calculation depending on project and standard name (#620) Valeriu Predoi
- State of the art FX variables handling without preprocessing (#557) Valeriu Predoi
- Add max, min and std operators to multimodel (#602) Javier Vegas-Regidor
- Added preprocessor to extract amplitude of cycles (#597) Manuel Schlund
- Overhaul concatenation and allow for correct concatenation of multiple overlapping datasets (#615) Valeriu Predoi
- Change volume stats to handle and output masked array result (#618) Valeriu Predoi
- Area_weights for cordex in area_statistics (#631) mwjury
- Accept cubes as input in multimodel (#637) sloosvel
- Make multimodel work correctly with yearly data (#677) Valeriu Predoi
- Optimize time weights in time preprocessor for climate statistics (#684) Valeriu Predoi
- Add percentiles to multi-model stats (#679) Peter Kalverla

CHAPTER

THIRTYFOUR

V2.0.0B9

This release includes

34.1 Bug fixes

- Cast dtype float32 to output from zonal and meridional area preprocessors (#581) Valeriu Predoi

34.2 Improvements

- Unpin on Python<3.8 for conda package (run) (#570) Valeriu Predoi
- Update pytest installation marker (#572) Bouwe Andela
- Remove vmrh2o (#573) Mattia Righi
- Restructure documentation (#575) Bouwe Andela
- Fix mask in land variables for CCSM4 (#579) Klaus Zimmermann
- Fix derive scripts wrt required method (#585) Klaus Zimmermann
- Check coordinates do not have repeated standard names (#558) Javier Vegas-Regidor
- Added derivation script for co2s (#587) Manuel Schlund
- Adapted custom co2s table to match CMIP6 version (#588) Manuel Schlund
- Increase version to v2.0.0b9 (#593) Bouwe Andela
- Add a method to save citation information (#402) SarahAlidoost

For older releases, see the release notes on https://github.com/ESMValGroup/ESMValCore/releases.

Part VIII

Indices and tables

- genindex
- search

PYTHON MODULE INDEX

e

INDEX

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

Z